GRAPH DECOMPOSITION ALGORITHMS FOR ANALYZING SOCIAL AND LARGE COMPLEX NETWORKS

WALI MOHAMMAD ABDULLAH
Master of Science, IICT, Bangladesh University of Engineering and Technology, 2014

A thesis submitted
in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

THEORETICAL AND COMPUTATIONAL SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

GRAPH DECOMPOSITION ALGORITHMS FOR ANALYZING SOCIAL AND LARGE COMPLEX NETWORKS

WALI MOHAMMAD ABDULLAH

Date of Defence: June 09, 2022

Dr. S. Hossain Supervisor	Professor	Ph.D.
Dr. R. Benkoczi Committee Member	Professor	Ph.D.
Dr. M. Khan Committee Member	Chief Tech Officer	Ph.D.
Dr. P. Ghazalian Internal External Examiner	Associate Professor	Ph.D.
Dr. W. Haque External Examiner University of Northern British Columbia, Prince George, BC	Professor	Ph.D.
Dr. W. Osborn Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

*To **my parents** for all their support and sacrifices*

*To **my wife** for being with me through the good and bad stuff*

*To **my son** for making my world beautiful*

Abstract

Graphs are often used to model or represent large and sparse networks with billions of vertices and edges and store extensive amounts of structural and semantic information. Therefore, analyzing characteristics in networked data, such as graphs that can yield important information on the modelled structure, is challenging due to their linked nature and size. A common way to uncover this high-quality information is by analyzing subgraphs to get a deeper understanding of the data, which are helpful for classification, clustering, and knowledge discovery. This thesis proposes using a compact network data representation based on sparse matrix data structures. We will consider the enumeration of subgraphs (edge clique cover problem) with some ordering schemes. Finally, we benefit from the linear algebraic approach to graph algorithms for counting triangles, triangle enumeration, the k-count algorithm, and triangle centrality calculation. This thesis will present both serial and parallel algorithms for solving these problems.

Acknowledgments

First and foremost, I would like to thank the Almighty Allah for giving me the opportunity, strength, and patience to undertake this research. This work would not have been possible without His blessing.

I am lucky that I have worked under the supervision of Dr. Shahadat Hossain. The way he treated me made me feel like he was not my supervisor but my guardian. It may be difficult for me to work under any supervisor after working with such a great person. Thank you, Sir, for everything.

I want to express my sincere gratitude to my supervisory committee members, Dr. Saurya Das, Dr. Robert Benkoczi, and Dr. Muhammad Ali Khan. Their guidance, encouragement, and suggestions helped me a lot. Their immense efforts and way of directing the students of the optimization research group can be a model to others.

Without the encouragement I got from my family, it would not be possible for me to come this far and go forward. I am very grateful to my parents: Dr. Md Mohiuddin Abdullah & Afsir Begum, my parents-in-law: Late Md Tazul Islam & Joly Akhter, sisters: Mahmuda & Shahnaz, brothers: Deen & Rafat. I want to thank my wife, Sharmin Islam. Without her, my life would be a lot more complicated.

I also want to thank all my friends, well-wishers and all the members of the Mathematics and Computer Science Department. Special thanks to David Awosoga for his collaborations during my research.

I am thankful to the Alberta Innovates for Technology Futures Graduate Student Scholarship, the School of Graduate Studies (SGS), Graduate Students' Association (GSA), and the Department of Mathematics and Computer Science for their financial support. A part

of our computations was performed on Compute Canada HPC system, and I gratefully acknowledge their support. I thank Lab2Market-Manitoba and their partners Mitacs and North Forge for providing me the opportunity for an internship during my Ph.D. studies.

Last but not least, I am also grateful to the researchers for their ideas and contributions in this field.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

We can store an extensive amount of information using graphs. From these graphs, one can extract lots of information by analyzing and mining. However, extracting information from large graphs is challenging due to their sparse nature. Therefore, we see significant interest in studying and mining data from extensive sparse graphs in the literature.

Analyzing graph properties is the most basic form of graph analysis. This analysis is related to graph dynamics, such as density, diameter, and clique number of a given graph. For instance, density measures the interconnectedness of the graph, diameter measures the distance among vertices in the worst or average case, and clique number corresponds to the size of the largest complete subgraph in the graph.

In most cases, graph properties are simple numeric values and can be calculated easily. Unfortunately, these properties do not give the full picture of the information that a graph can contain. For instance, Figure 1.1 shows that two graphs that are structurally different can have the same number of vertices and edges. There are different ways for graph decomposition, but all of them are useful in large network analysis. Now, if we consider the number of complete subgraphs of a graph, then the two graphs shown in Figure 1.1 give two different numbers. Identification of special interest groups or characterization of information propagation are examples of frequently performed operations in social networks [82] which we can examine through graph analysis.

Analyzing characteristics in networked data, such as graphs can yield important infor-

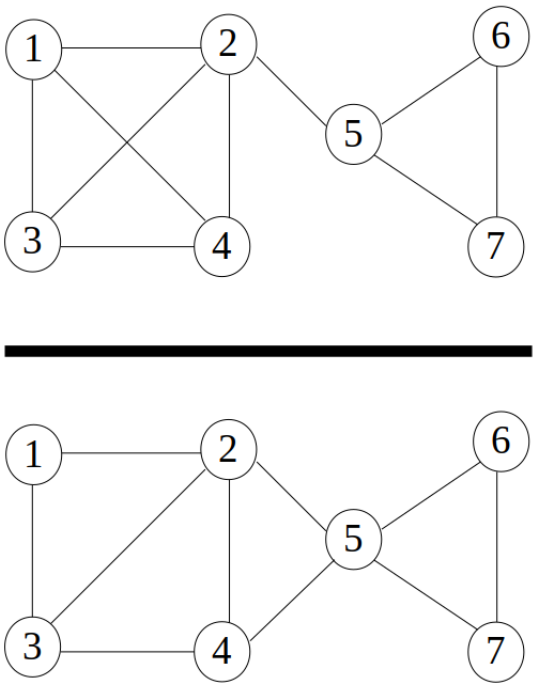


Figure 1.1: Example of two different graphs having the same number of vertices and edges.

mation on the structure being modelled. Identification of, and computation with, dense or otherwise highly connected subgraphs are two of the kernel operations arising in areas as diverse as sparse matrix determination, and complex network analysis [43, 36, 40]. Efficient representation of network data is critical to addressing algorithmic challenges in analyzing massive data sets using graph-theoretic abstractions. In particular, enumerating graph is a way to discover essential features in graphs and compactly represent sparse graphs. Analyzing graph (using decomposition or enumeration) has different methods. For instance, one of the methods can be finding a set of k cliques that cover all the edges of the given graph. This problem is called the edge clique cover problem (see Figure 2.9). Another problem could be counting and enumerating local topological structures, such as triangles.

We see an enormous number of algorithms for graph analysis from the literature, especially clique cover, triangle count & enumeration, k-count, and triangle centrality. Coen Bron and Joep Kerbosch [7] presented an exact algorithm for generating all the maximal cliques of a graph. Again, Etsuji Tomita et al. [81] presented a depth-first search algorithm

to solve this problem. Gramm [30] shows exact algorithms are feasible; for instance, classes that can be covered with a small number of cliques. Some researchers solved this problem for sparse graphs [25]. Erich Prisner [64] showed that for certain superclasses of the class of line graphs, their presented algorithm could compute minimum sets of maximal cliques covering and partitioning the edge set of a graph efficiently. Counting and enumerating triangles from a given graph is another form of graph analysis, where we can say a triangle is a clique with cardinality three. Hasan and Dave [2] discussed the existing methods of triangle counting, ranging from sequential to parallel, single-machine to distributed, exact to approximate, and off-line to streaming. Li and Bader [51] presented a rapid implementation of triangle centrality using SuiteSparse GraphBLAS.

Large real-life networks are usually sparse and are challenging to analyze. In the literature, we also see that the graph algorithms can be expressed in the language of linear algebra by exploiting this duality between graphs and sparse matrices [43, 44]. Graph kernel operations such as breadth-first search, graph connected components etc. can be expressed as sparse matrix-matrix multiplication [43]. This mapping between graphs and sparse matrices can be exploited in data analysis and algorithm design. This is especially important because of the unstructured nature of data access and the memory-bound nature of graph algorithms. From the inception of the graph theory, the duality between the graph representation as abstract collections of vertices and edges and a sparse adjacency matrix representation has been a part of graph decomposition.

In mathematics, a duality translates concepts, theorems, or mathematical structures into other concepts, theorems, or structures, in a one-to-one fashion. The adjacency matrix A in Figure 1.2 is dual with the corresponding undirected graph $G = (V, E)$, then the vector-matrix multiply is dual with breadth-first search, where we can find the neighbor of vertex 4, i.e. 3,5,6, and 7. Adjacency matrices (2D array) have not traditionally been used for analyzing large sparse graphs. Therefore, we need efficient data structures and algorithms for the practical array-based approach to computing large sparse networks (graphs).

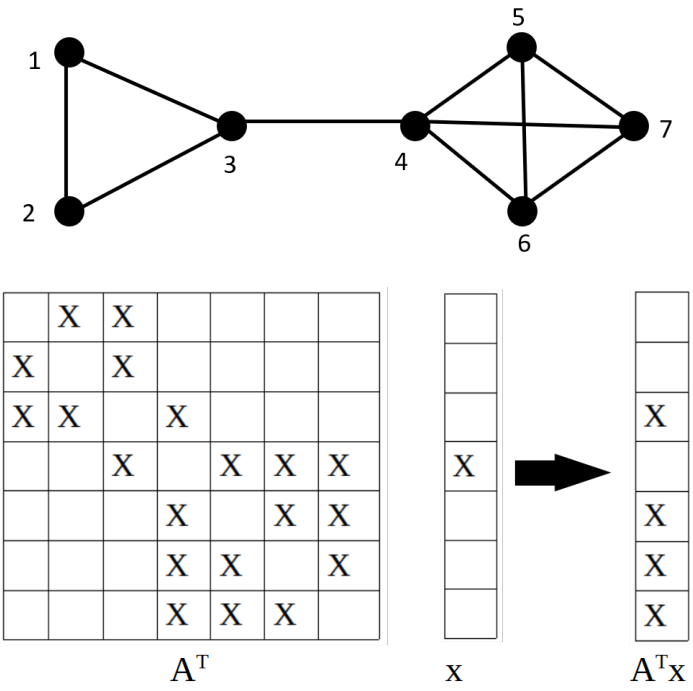


Figure 1.2: (a) An undirected graph G, (b) Matrix graph duality

In this thesis, we benefit from the linear algebraic approach to graph algorithms to analyze graphs and, more precisely, solve edge clique cover problems, count & enumerating triangles for a given graph, and calculate k-count and triangle centrality. Basing simple but crucial observations on some linear algebraic approaches, we provide improved algorithms for these problems.

While there have been excellent and continuing developments in algorithm theory, the algorithms are usually designed and analyzed under the assumption of some abstract machine model, such as RAM. In theoretical computer science, efficiency usually means polynomial-time solvability. Unfortunately, there has been a growing gap between algorithm theory and its effective implementation. Consequently, many excellent theoretical algorithmics rarely translates into practical implementation. In this thesis, one of our key goals is to fill this gap by engineering effective data representation to design algorithms that scale to big-data instances [58].

1.1.1 Some Applications

Computational Biology

In computational biology, discovering protein complexes is a well-known problem. A protein complex is a group of proteins that bind together to perform a specific task. Multi-protein complexes carry out most cellular processes. At different stages of a cellular process, proteins form stable complexes, and some others form transient associations [84]. To analyze evolutionary mechanisms behind biological networks, finding the organization of proteins into overlapping complexes is essential. Now, these protein networks can be represented using graphs where the proteins would be the vertices, and edges would represent the interactions between proteins. Therefore, analyzing graphs can help to find these protein interactions.

Food Science

In food science, one interesting problem is to efficiently analyze large numbers of combinations of food items, such as components on a salad [59]. For instance, we can use a graph to represent the interaction between the food components, where components are the vertices and the edges' compatibility. Using any graph analytic method, we can find a set of food components perfect for a specific salad.

Efficient Representation of Pairwise Information

In different studies, we see that the representation of pairwise information is crucial because of the comparison required between two products for their statistical difference, equivalence, and compatibility [21]. Therefore, analyzing graphs can help find these comparison results by efficiently representing them. For instance, if there are many products with many attributes, the total number of comparisons can be tens of thousands.

Maximal Cliques

Community detection from a network is another well-studied problem. Dense subgraphs are often used to detect communities in undirected and connected graphs $G = (V, E)$. Identification of, and computation with, dense or otherwise highly connected subgraphs are two of the kernel operations arising in areas as diverse as sparse matrix determination, and complex network analysis [43, 36, 40]. Identification of special interest groups or characterization of information propagation are examples of frequently performed operations in social networks [82]. Therefore, finding maximal cliques or covering edges by a set of subgraphs can still be seen in future research.

1.1.2 Use in Real Large Complex Networks

Analysis of real-world networks has attracted significant attention due to its enormous real-life applications. In the literature, we see many algorithms addressing this problem. However, different contexts sometimes require different algorithms: for example, understanding network evolution over time can be found from triangle counting and detecting community using clique cover. Because of the size of these networks, some algorithms cannot manage them due to insufficient memories and long running times. However, these real-world networks are sometimes relatively easy to process, and in that case, the primary issue would be storing all the network (graph) information efficiently. Schmidt et al. [76] showed how real-world networks contain the number of maximal cliques, which is sometimes linear in the size of the graph where the number of maximal cliques in a graph could be exponential. Therefore, instead of general real-world network analysis, we should look at the problem from different angles, leading us to an efficient algorithm.

1.2 Content of the Thesis and Contributions

- **Chapter 2.** We start Chapter 2 discussing the graph theoretic concepts that we use in the rest of this thesis. First, we discuss the theoretical progress that has been made

on the clique cover problems and the practical design of algorithms. We then discuss some of the recent exact and heuristic algorithms. The reviews and summaries of this chapter give a broad idea of the background of this thesis.

- **Chapter 3.** Efficient representation of network data is critical to addressing algorithmic challenges in analyzing massive data sets using graph-theoretic abstractions. In this chapter, we propose sparse-matrix data structures to enable the compact representation of graph data and use an existing sparse matrix framework [32] to design efficient algorithms for the Edge Clique Cover (ECC) problem.

  We see many algorithms in the literature solving the ECC problem, but only a few exact methods because of the limitation to solve small-size instances. In a recent approach, Gramm et al. [27] introduced and analyzed data reduction techniques to shrink the instance size without sacrificing the optimal solution. Their main idea was about the feasibility of exact algorithms for small enough instances.

  In this chapter, we present a compact representation of network data based on sparse matrix data structures [35]. Our implemented algorithm is based on an algorithm due to Kellerman [42]. We experimentally verify that the ECC algorithm is sensitive to the ordering in which the vertices are processed. We employ three vertex ordering algorithms from the literature: Largest-degree order (LDO), Degeneracy order (DGO), and Incidence-degree Order (IDO) prior to applying the algorithm [42].

  Part of the work in Chapter 3 is published in the following contributions.

  - Wali M. Abdullah, Shahadat Hossain, and Muhammad A. Khan. *Covering Large Complex Networks by Cliques - A Sparse Matrix Approach*. AMMCS 2019 International Conference, Waterloo, Canada, 2019.
  - Wali Mohammad Abdullah, Shahadat Hossain, and Muhammad Ali Khan. *Covering large complex networks by cliques—a sparse matrix approach*. in: D. M. Kilgour, H. Kunze, R. Makarov R. Melnik, X. Wang (Eds.), Recent De-

velopments in Mathematical, Statistical and Computational Sciences, Springer International Publishing, Cham, 2021, pp. 117–127.

- **Chapter 4.** The *Edge Clique Cover problem (ECC)* considered in Chapter 4 is concerned with finding a collection of complete subgraphs or cliques such that every edge and every vertex of the input graph is included in some clique. The computational challenge is to find an ECC with the smallest number of cliques (*minECC*). The minECC problem is computationally intractable or NP-hard [47].

  Effective representation of network data is critical to meeting algorithmic challenges for exactly or approximately solving intractable problems, especially when the instance sizes are large and sparse. We use sparse matrix data structures to enable compact representation of sparse network data and an existing sparse matrix framework [32] to design efficient algorithms for the minECC problem motivated by the works of Bron et al. [7], and E. Tomita et al. [81].

  In Chapter 3, we used a similar compact representation of network data. While the vertex-centric ECC algorithm from Chapter 3 frequently produced smaller clique covers compared with other methods, the high memory footprint of the method made it less scalable on very large problem instances. In this chapter, we propose an "edge-centric" minECC algorithm, which exhibits good scalability when applied to extremely large synthetic and real-life network instances.

  We present the comparing results concerning the clique cover size and runtime with the current state-of-the-art algorithm for minECC [17]. For 219 test instances (from DIMACS10, SNAP, Real-World, Small-World, and Erdős-Rényi groups), where the number of edges varies between 170 and 7.6×10^7, our "edge-centric" minECC algorithm produces smaller or equal size clique covers than the algorithm presented in [17]. Our "edge-centric" minECC algorithm is also significantly faster than state-of-the-art algorithm for minECC [17]. Finally, we present a post-processing step to get

the assignment minimum edge clique cover motivated by Ennis et al. [22].

In Section 4.5, we present a parallel version of our "edge-centric" minECC algorithm. We compile and run the code using OpenCilk's built-in LLVM, Clang, and Clang++ with optimization level 3 (O3 flag). Our algorithm shows promising speedup for large test instances while we varied the number of threads between 1 and 8.

Part of the work in Chapter 4 is published as an abstract, a short paper, and a refereed contribution.

- Wali Mohammad Abdullah, Shahadat Hossain, and Muhammad Ali Khan. *A Sparse Matrix Approach for Covering Large Complex Networks by Cliques*. 18th Cologne-Twente Workshop on Graphs and Combinatorial Optimization. Sep 2020.
- Wali Mohammad Abdullah, Shahadat Hossain, and Muhammad Ali Khan. *A Sparse Matrix Approach for Covering Large Complex Networks by Cliques*. Canadian Operational Research Society (CORS-2021). June 2021.
- Wali Mohammad Abdullah, and Shahadat Hossain. *A Sparse Matrix Approach for Covering Large Complex Networks by Cliques*. In International Conference on Computational Science. London, UK. Springer. 2022.
- Part of this chapter is prepared to submit in a journal.

- **Chapter 5.** Triangles are an essential part of network analysis, representing metrics such as transitivity and clustering coefficient. Linear algebraic methods can use the correspondence between sparse adjacency matrices and graphs that can apply for triangle counting and enumeration. Here, the main computational kernel is sparse matrix-matrix multiplication.

  In this chapter, we use an intersection representation of graph data implemented as a sparse matrix and engineer an algorithm to compute the "k-count" distribution of

the triangles of the graph. The main computational task of computing sparse matrix-vector products is carefully crafted by employing compressed vectors as accumulators. Our method avoids redundant work by counting and enumerating each triangle exactly once. We present results from extensive computational experiments on large-scale real-world and synthetic graph instances that demonstrate the excellent scalability of our method. In terms of run-time performance, our algorithm has been found to be orders of magnitude faster than the reference implementations of the **miniTri** data analytics application [83].

In Section 5.4, we describe the parallel implementation of our intersection algorithm, **fullCount** for triangle count, enumeration, k-count, and triangle centrality. First, we describe the parallel triangle counting algorithm. Then we discuss the parallel algorithm for our **fullCount** algorithm concerning triangle count, local triangle count, and k-count. Finally, we present a parallel version of our triangle centrality calculation method. A shared memory parallel implementation of our algorithm using OpenMP is being developed. We observe reasonable speedups using multiple threads.

Part of the work in Chapter 5 is published in the following refereed contributions.

- Wali Mohammad Abdullah, David Awosoga, and Shahadat Hossain. *Intersection Representation of Big Data Networks and Triangle Counting*. In 2021 IEEE International Conference on Big Data (Big Data), pages 5836–5838, 2021, and
- Wali Mohammad Abdullah, David Awosoga, and Shahadat Hossain. *Intersection Representation of Big Data Networks and Triangle Enumeration*. In International Conference on Computational Science. London, UK. Springer. 2022.

- **Chapter 6.** We conclude this thesis by summarizing the ideas and results of all the chapters and pointing out some future directions of our research.

Chapter 2

Background

2.1 Introduction

Graphs are among the essential abstract data structures in computer science, and the algorithms that operate on them are critical to modern life. The field of graph algorithms has become one of the pillars of theoretical computer science, informing research in such diverse areas as combinatorial optimization, complexity theory, and topology. In our large-network analysis, the simplicity and generality of graphs play potent tools in modeling our problems.

This chapter discusses the graph theoretic concepts that we use in this chapter and the rest of this thesis. First, we discuss the theoretical progress that has been made on the clique cover problems and the practical design of algorithms. We then discuss some of the recent exact and heuristic algorithms.

2.2 Preliminaries

2.2.1 Graph

A graph $G = (V,E)$ consists of a set of vertices V, and a set of edges $E \subseteq V \times V$, which represent relationships (or links) between two vertices. **For rest of this thesis we let $|E| = m$ be the number of edges and $|V| = n$ be the number of vertices.**

Kozyrev and Yushmanov [48] surveyed representation of graphs by families of sets of objects of different kinds, such as, intersection graphs of geometrical objects, curve graphs, function graphs, permutation graphs, circular permutation graphs, chord graphs, circular-

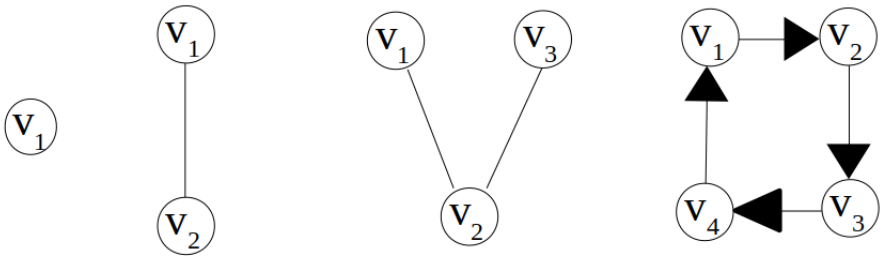


Figure 2.1: Example of different graphs.

arc graphs, interval graphs, unbounded interval graphs, intersection graphs of subgraphs of a graph, chordal graphs, split graphs, 3-split graphs, undirected path graphs, directed path graphs. In this section, we only discuss some common and related graphs.

Undirected Graph

An undirected graph is a graph whose edges are unordered pairs of vertices and do not have a direction. Sometime these undirected graphs are called undirected networks where all the edges are bidirectional.

From now on, we will use "$G = (V, E)$" to mean "an undirected and connected graph" unless explicitly stated otherwise.

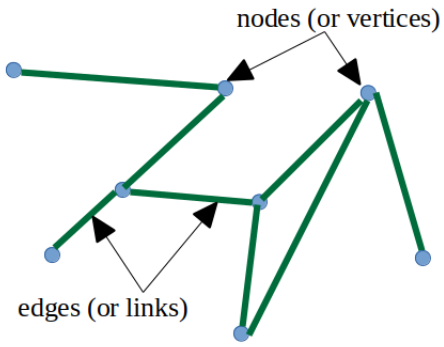


Figure 2.2: Example of an undirected graph.

Directed Graph

A directed graph (or digraph) is a graph whose edges (or arcs) are ordered pairs of vertices and edges have direction. Figure 2.3 shows a simple directed graph, where the

double-headed arrow represents two distinct edges, one for each direction.

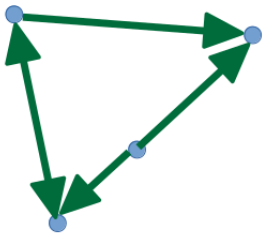


Figure 2.3: Example of a simple directed graph.

Subgraph

Let $G = (V, E)$ be a graph. A graph, G', is called subgraph of G, if the vertices and edges of G' are subsets of V and E of G respectively. Formally we can define as follows.

A graph $G' = (V', E')$ is a subgraph of another graph $G = (V, E)$ iff

- $V' \subseteq V$, and $E' \subseteq E$

An induced subgraph of a graph G is a graph formed from a subset of V and all of the edges connecting pairs of vertices in that subset.

Interval Graph

A graph is an interval graph if and only if each of its vertices can be associated with an interval on the real line in such a way that two vertices are adjacent in the graph exactly when the corresponding intervals have a nonempty intersection.

Let a family of intervals are given as S_i, where $i = 1, 2, \ldots, n$. Then, we can define an interval graph formally as follows. An interval graph is an undirected graph $G = (V, E)$ formed from S by creating one vertex v_i for each interval S_i, and $E = \left\{ \{v_i, v_j\} | S_i \cap S_j \neq \emptyset \right\}$.

Competition Graph

Biologists describe the feeding relations among species living together in a community by a "food web," a directed graph with vertices corresponding to species and a directed

edge from a to b if b preys on a (energy flows from a to b) [13].

Let $D = (V, E_D)$ be a directed graph (which represents a food web), where V is the set of vertices and E_D is the set of directed edges or arcs. A competition graph of D is an undirected graph $C(D)$, and we say two vertices i and j are in competition if there are arcs $\{l, i\}$ and $\{l, j\}$ in E_D. Then there will be an edge $\{i, j\}$ in graph $C(D)$. For example, Figure 2.5 shows a competition graph for a given digraph shown in Figure 2.4.

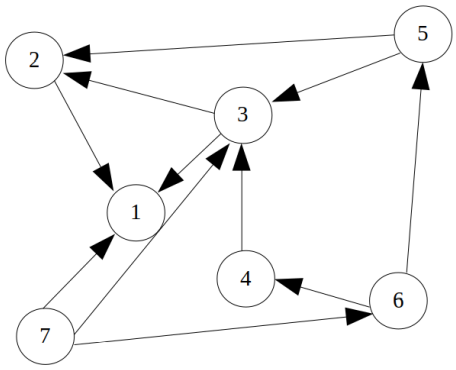


Figure 2.4: Example of a digraph D

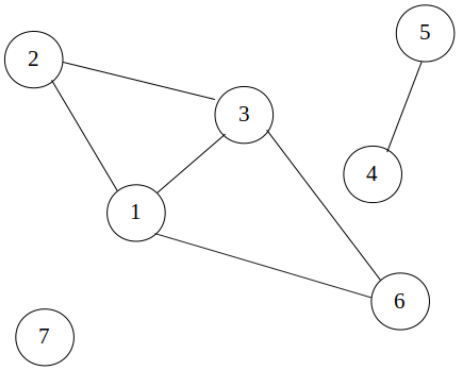


Figure 2.5: The competition graph of the digraph shown in Figure 2.4

Niche Graph

Let $D = (V, E_D)$ be a directed graph (Figure 2.4) which represents a food web and $C(D)$ be the competition graph of D (Figure 2.5). Now, two vertices p and q, where $p, q \in V$, have common enemy if there are arcs $\{p, r\}$ and $\{q, r\}$. Therefore there will be an edge $\{p, q\}$ in common enemy graph $CE(D)$ (Figure 2.6).

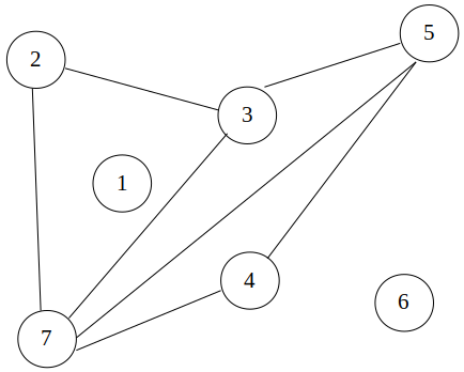


Figure 2.6: The common enemy graph of the digraph shown in Figure 2.4

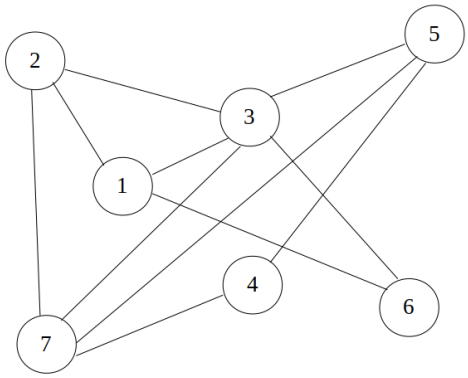


Figure 2.7: The niche graph of the digraph shown in Figure 2.4

The niche graph corresponding to D (i.e. $N(D)$) is the undirected graph $G = (V,E)$ with an edge between two distinct vertices x and y of V if and only if for some $z \in V$, there are arcs $\{x,z\}$ and $\{y,z\}$ in E_D or there are arcs $\{z,x\}$ and $\{z,y\}$ in E_D. Figure 2.7 is the niche graph of D. Here, an edge between two vertices represents they have a common enemy, a common prey or both.

2.2.2 Set Representation

Consider the undirected simple graph G which has six vertices and nine edges as shown in Figure 2.8. Graph G can be represented by a family of finite sets. $S_1 = \{e_1, e_2, e_3, e_4\}$, $S_2 = \{e_1, e_5, e_6, e_7\}$, $S_3 = \{e_2, e_5, e_8, e_9\}$, $S_4 = \{e_3, e_6\}$, $S_5 = \{e_4, e_8\}$ and $S_6 = \{e_7, e_9\}$. Here, S_i represents the set of edges incident on vertex i. Let, the family of subsets be represented by $F = \{S_1, \ldots, S_6\}$.

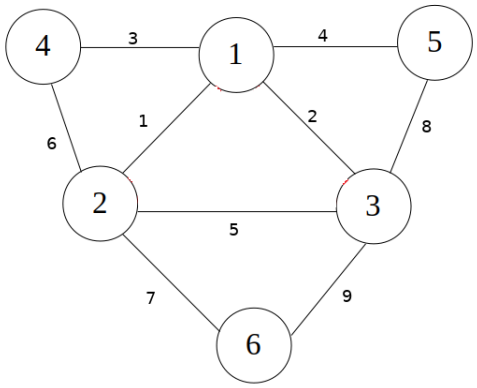


Figure 2.8: A simple graph G

k-set Representation

A family of finite set, F is called a k-set representation of G if each of the subset S_i has at least k elements.

Distinct Set Representation

We say F is a distinct set representation of graph G if S_i and S_j do not contain exactly the same elements, i.e. $S_i \neq S_j$ for all $i, j \in V$.

Simple Set Representation

We have a simple set representation if the intersection between two different subsets is less than or equal to one, i.e. $|S_i \cap S_j| \leq 1$. In other wards, the given graph G does not contain any multiple edges.

2.2.3 Clique

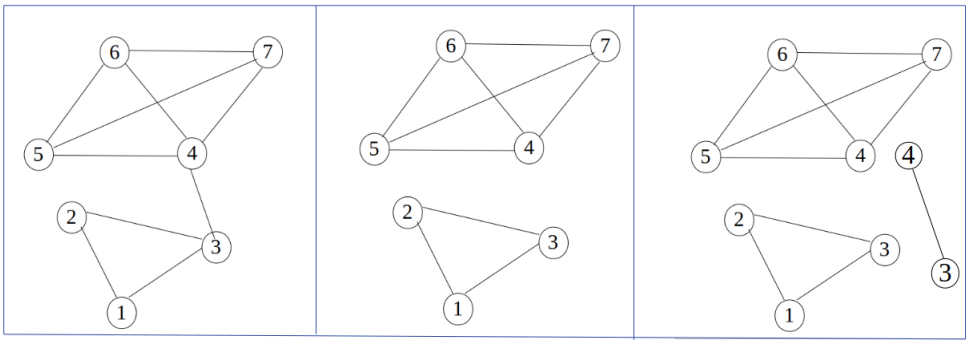
A clique is a subset of vertices such that every pair of distinct vertices are connected by an edge in the induced (by the subset of vertices) subgraph. In other words, a clique of an undirected and connected graph G is a complete subgraph of G.

Vertex Clique Cover (VCC)

A vertex clique cover is a set of cliques that cover all the vertices of a graph. If two cliques share the same vertex v, we can delete v from one of the two cliques. The vertex clique cover number is the smallest number of cliques needed to cover all the vertices.

Edge Clique Cover (ECC)

An edge clique cover of size k in graph G is a decomposition of set E into k subsets $C_1, C_2, \ldots, C_k$ such that $C_i, i = 1, 2, \ldots, k$ induces a clique in G and each edge $\{u, v\} \in E$ is included in some C_i. In this thesis, we will focus on the edge clique cover problem. **From now on, we will use "clique cover" to mean "edge clique cover" unless explicitly stated otherwise.**



a) G=(V,E), |V|=7, |E|=10 b) VCC (2-cliques) c) ECC (3-cliques)

Figure 2.9: Example of clique covers

Clique Partition

A clique partition of G is a set of cliques of G, containing each edge of G exactly once. We can also refer to this as edge clique partition. **From now on, we will use "clique partition" to mean "edge clique partition" unless explicitly stated otherwise.**

Clique Cover Number ($cc(G)$)

The smallest cardinality of any clique covering of G is called the clique covering number of G and is denoted by $cc(G)$. $cc(G) = \min \{|C| : C \text{ a clique cover of } G\}$

Trivial Clique Cover

A trivial clique cover can be specified by the set of edges E with each edge being a clique.

Minimum Clique Covering

The minimum clique cover of G is a clique cover C with $|C| = cc(G)$

Clique Partition Number ($cp(G)$)

The smallest cardinality of any clique partition of G is called the clique partition number of G and is denoted by $cp(G)$. $cp(G) =$ min $\{|P| : P$ a clique partition of $G\}$

Minimum Clique Partition

The minimum clique partition of G is a clique partition P with $|P| = cp(G)$.

2.3 Literature Review

Edge clique covering of a simple undirected and connected graph with a minimum number of complete subgraphs (cliques) is a well-studied problem. We have already indicated earlier that a closely related problem is vertex clique cover, which can also be formulated as a coloring problem. The edge clique covering problem was widely studied under a few different names: Covering by Cliques, Intersection Graph Basis, and Keyword Conflict. This chapter reviews some of the central results from the literature on the edge clique cover problem. This review also emphasizes different applied problems related to the edge clique cover problem.

N.J. Pullman [65], and Roberts [67] surveyed some of the progress on clique covering and partitioning where the works concerned on global bounds on clique partition number and clique cover number. Let, k be the number of cliques to cover all the edges of the given graph G. A "bipartite graph" is a graph whose vertices can be divided into two disjoint and independent sets V_1 and V_2 such that every edge connects a vertex in V_1 to one in V_2.

A complete bipartite graph K is a special kind of bipartite graph where every vertex of the first set V_1 is connected to every vertex of the second set V_2. Erdös et al. [26] presented the following theorem.

Theorem 2.1. *If G is any graph with n vertices, then* $cp(G) \leq \lfloor n^2/4 \rfloor$ *with equality holding for* $G = K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$ *[26]*

We note that if $C = \{C_1, C_2, \ldots, C_k\}$ is a k-clique partition for a graph then it is also a k-clique cover for the same graph. Therefore, $cc(G) \leq cp(G)$. Thus, the upper bound on $cp(G)$ provided by the Theorem 2.1 is also an upper bound on $cc(G)$.

For general graphs, not much is known on the relationship between $cc(G)$ and $cp(G)$. If G is a complete graph on n vertices we $cc(G) = cp(G) = 1$ and therefor $1 \leq cc(G) \leq cp(G) \leq \frac{n^2}{4}$. Erdös et al. show that for large n, $cp(G)/cc(G) \leq \frac{n^2}{12}$. Erdös et al. [26] also mentioned a possible improvement in the bound if we know the number of edges of the graph. Later Lovász [52] presented Theorem 2.2 to address that improvement.

Theorem 2.2. *Let* $G = (V, E)$ *is any graph with* $|V| = n$ *vertices and* $|E| = m$ *edges. If* $k = \binom{n}{2} - m$ *and* $t = max\{s \in \mathbb{Z} : s^2 - s \leq k\}$*, then* $cc(G) \leq k + t$*, where* $\mathbb{Z}$ *is the set of nutural numbers [52]*

The computational intractability of edge clique cover was independently proved by Alon [39] and Kou et al. [47]. Orlin established the NP-hardness of the minimum edge clique cover problem (ECC).

Theorem 2.3. *The problem of deciding if cc(G) is at most k is NP-complete [47]*

Chung and Muller showed that ECC remains NP-hard even when the graph is planar, while Hoover showed that ECC is NP-hard for graphs with a maximum vertex degree of six. On the other hand, ECC can be solved in polynomial time when restricted to chordal graphs, line graphs, and circular-arc graphs. ECC is one of the hardest computationally intractable problems; it is not approximable within a factor of $|V|^{\varepsilon}$ for some $\varepsilon > 0$, unless $P = NP$ [54].

We see some interesting theoretical developments for special graphs, such as interval, competition, niche, and line graphs. For these graphs, the authors derive some better bounds or favorable complexity results.

Roberts shows the connection between the intersection number and edge clique cover [67]. There are many other authors who also have observed the same result [26], [47], [60], [5], [63], [78].

Theorem 2.4. *For all graph G, $IN(G) = cc(G)$, where $cc(G)$ is the minimum edge clique cover number [67]*

Cohen [13] showed that the theory of "interval graph" can illuminate the study of ecological systems mentioning two problems concerning the structure of "ecological phase space" and "the food web". He also introduced the concept of the "competition graph" in connection with this study. From a food web, $D = (V, E_D)$, we can get a competition graph, by which one can identify competition between vertices (species).

The competition number $\kappa(G)$ was first introduced by Roberts in [66] for recognizing the competition graph. For any graph G, G together with sufficiently many isolated vertices is the competition graph of some acyclic digraph. Roberts defined the competition number $\kappa(G)$ of a graph G as the minimum number of such isolated vertices. Then Opsut [60] gave lower bounds for the competition number of graph G. Opsut settled a question posed by Roberts [66] by demonstrating that the problem of determining whether or not an arbitrary graph is the competition graph of some acyclic digraph is NP-complete. Sano [74] then generalized the lower bound for the competition number of a graph presented by Opsut. The new lower bound given by Sano is a reliable tool to compute the exact values of the competition numbers of graphs.

Cable et al. [10] presented the following theorem relating the niche graph with the edge clique cover.

Theorem 2.5. *G is a niche graph if and only if G has subgraphs G_1 and G_2, $G = G_1 \cup G_2$,*

and G_1 has an edge clique cover $\{C_1,\ldots,C_k\}$ such that $i \in C_j$ implies $i > j$ and $\{i,l\}$ is an edge in G_2 if and only if $C_i \cap C_l \neq \emptyset$ [10]

Sean and Rolf [55] studied the number of distinct minimal clique partition and clique covers of line graphs. Let G be an undirected graph. H is called the line graph of G if the vertices of H are the edges of G, and two vertices x and y in H are adjacent if and only if (viewed as edges in G) they intersect. $G^\star$ also denotes the line graph of G. A wing in G is a triangle with the property that exactly two of its vertices have degree 2 in G. Authors obtained a constructive proof of Orlin's result [61], where Orlin determined the clique covering and clique partition numbers $cc(G^\star)$ and $cp(G^\star)$ respectively (Theorem 2.6).

Theorem 2.6. *Let G be an undirected graph, $G \neq K_3$, and let v_2 be the number of vertices of degree at least two in G and w be the number of wings in G. Then $cc(G^\star) = v_2 - w$ and $cp(G^\star) = v_2$. [61]*

In terms of easily calculable parameters of G, Sean and Rolf [55] were able to completely enumerate the number of distinct minimal clique covers and partitions of G using that constructive proof.

An *isolated vertex v* is a vertex with degree zero. Let, an undirected and connected graph $G = (V,E)$, where V is the set of vertices and E is the set of edges. Vertices x and y ($x,y \in V$) are *equivalent*, if $\{x,y\} \in E$ and for all vertices z ($z \in V$) different from x and y, $\{z,x\} \in E$ if and only if $\{z,y\} \in E$. (Figure 2.10). András Gyárfás [31] presented a lower bound on edge coverings by cliques for graphs with no isolated vertices and no equivalent vertices.

Let $cc(G)$ denotes the edge clique-cover number of a graph G.

Theorem 2.7. *If a graph G has n vertices and G contains neither isolated vertices nor equivalent vertices then $cc(G) \geq log_2(n+1)$ [31]*

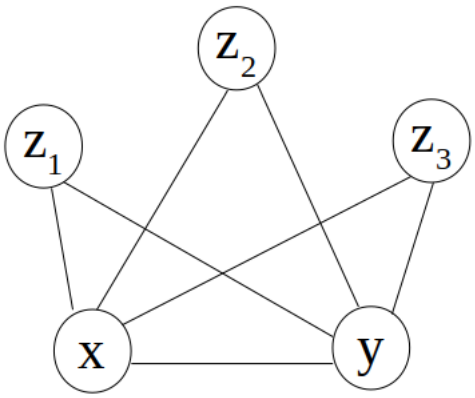


Figure 2.10: Equivalent vertices (x, y)

Determining $cc(G)$ for an arbitrary graph is NP-hard, and much progress has been made in producing algorithms to find such coverings. We see three different methodologies that have been used to solve edge clique cover problems; these are branch and bound, integer linear programming, and data reduction. Besides these exact algorithms, we also see heuristics to solve edge clique cover problem.

In 1973, Bron and Kerbosch [7] proposed an algorithm to list all the maximal cliques of a given graph using a branch-and-bound technique. Cutting off branches of the search tree that will not lead to new cliques at a very early stage the algorithm attempts to minimize the computation. In order to allow an easy understanding, we are going to discuss their algorithm in two steps. First, we discuss the notations, basic principles, and data structure used for the algorithm, and then we discuss the Bron-Kerbosch algorithm.

A maximal clique (complete subgraph) is a clique that cannot be extended by including one more adjacent vertex, meaning it is not a subset of a larger clique.

For the Bron-Kerbosch algorithm, the following three sets of vertices play a significant role.

- The set *CS* (Complete Subgraph) is the set of vertices that induces a complete subgraph of G. The set *CS* is extended by one vertex on branching or reduced by one vertex on backtracking during the execution of the algorithm.

- The set *CA* is a set of vertices, where the vertices will be used to extend *CS* towards a maximal clique. Therefore these vertices are referred to as candidates.
- The set *NOT* contains all vertices that were previously used to extend *CS* and are now explicitly excluded from the extension.

The two sets *CA* and *NOT* contain all vertices not contained in *CS* but adjacent to all vertices in *CS*. At recursion depth i, the sets *CA* and *NOT* are denoted by CA_i and NOT_i , respectively.

Bron-Kerbosch algorithm: The core of the algorithm is a recursively defined extension operator that uses the three sets described above. A call of the operator generates all extensions of the current set *CS* by adding vertices that are present in *CA* but not in *NOT*. We exclude the vertices from *NOT* because those vertices have already been considered at an earlier algorithm stage. Initially, the sets *CS* and NOT_0 are set to empty sets and CA_0 to V . Then, at recursion depth i, the extension operator performs the following five steps.

- (Step 1-i): Determine the vertex v from $NOT \cup CA$ with the least number of non-adjacent vertices in *CA*.
- (Step 1-ii): If v was found to be in *CA*, we take v as the next candidate. On backtracking of the extension operator, v is moved to *NOT*.
- (Step 2): Add v to *CS*.
- (Step 3): Create new sets CA_{i+1} and NOT_{i+1} from the old sets CA_i and NOT_i by removing all vertices not adjacent to v, keeping the old sets intact.
- (Step 4): Call the extension operator to operate on the sets *CS*, CA_{i+1}, and NOT_{i+1}.
- (Step 5): Upon return, remove v from *CS* and add it to NOT_i. Go to (Step 1).

Etsuji Tomita et al. [81] presented a depth-first search algorithm for generating all maximal cliques of an undirected graph. The main contribution of Tomita et al. is finding the

worst-case time complexity for generating all maximal cliques. The run time complexity of the algorithm is $O(3^{n/3})$, where n is the number of vertices.

Eppstein and Strash [25] implemented a new algorithm (due to Eppstein et al. [24]) for listing all maximal cliques in sparse graphs and analyzed its performance on a large corpus of real-world graphs. The algorithm presented by Tomita et al. [81] was faster than all other algorithms; however, the algorithm used an adjacency matrix of the input graph and required too much space for large sparse graphs. For sparse graphs, the algorithm presented by Eppstein and Strashin [25] is as fast or faster than Tomita et al., and sometimes faster by very large factors. For non-sparse graphs, their algorithm is sometimes slower than Tomita et al. but remains within a small constant factor of its performance. The implementation idea is described below.

Let, $G = (V,E)$ be an undirected graph with n vertices and m edges. For a vertex $v \in V$, let $\Gamma(v)$ be its neighborhood $\big\{w|\{v,w\} \in E\big\}$, and similarly, for a subset $W \subset V$, let $\Gamma(W)$ be the set $\cap_{w\in W}\Gamma(w)$, the common neighborhood of all vertices in W.

- The basic recursive Bron-Kerbosch algorithm maintaining three sets of vertices.

  1) a partial clique R,

  2) a set of candidates for clique expansion P, and

  3) a set of forbidden vertices Y.

  In each recursive call of this algorithm, a vertex v from P is added to the partial clique R. Then the sets of candidates for expansion and forbidden vertices are restricted to include only neighbors of v. Finally, if $P \cup Y$ becomes empty, the algorithm reports R as a maximal clique. On the other hand, if P becomes empty while Y is nonempty, the algorithm backtracks.

- With this basic idea, the pivoting heuristic reduces the number of recursive calls by choosing a vertex v in $P \cup X$ called a pivot. Tomita et al. [81] chose the pivot so that v has the maximum number of neighbors in P and, therefore, the minimum number of non-neighbors among all possible pivots.

- The algorithm of Eppstein et al. [24], slightly modified the previous algorithm. First, this algorithm orders all the vertices (degeneracy ordering). In this order, the recursive algorithm selects the vertices v to be used in each recursive call. Then for each vertex v in the order, it calls Tomita's algorithm to compute all cliques containing v and v's later neighbors while avoiding v's earlier neighbors.

In many studies, data reduction techniques for exactly solving NP-hard combinatorial optimization problems have proved helpful [29]. Preprocessing data (e.g., to eliminate redundancies) is a common practice used in many optimization solvers. One of the main objectives of preprocessing is to enable the main algorithmic task to execute faster or make the overall computational work more efficient. A specific instance of such a preprocessing framework, the polynomial time data reduction or kernelization is concerned with the reduction of the input size while preserving the optimal solution of the problem. The parameterized complexity theory aids in the rigorous analysis of data reduction in terms of a parameter that is independent of the input size of NP-hard problems. The time complexity of such kernelization algorithms can be expressed in terms of a function of the parameter and, therefore, independent of the input size. Gramm et al. [27] developed polynomial-time data reduction rules that, combined with a search tree algorithm (exact algorithm), allow for exact problem solutions.

Formally, an instance x of a parameterized problem comes with an integer parameter k. Where a parameterized problem is a language $L \subseteq \sum^* \times \mathbb{N}$ and $\sum$ is a finite alphabet. We say that a problem is fixed-parameter tractable (FPT) if there exists an algorithm solving any instance (x,k) in time $f(k)\,.\,|x|^{O(1)}$ for some computable function f and f is solely depending on parameter k. Here, x is the size of the input and k is the parameter.

Gramm et al. [27] proved the fixed-parameter tractability of covering edges by cliques. More specifically, clique cover can be solved in $O(f(2^k)+n^4)$ time, where n is the number of vertices of the given graph.

A closely related problem is deciding whether the edge-set of a given graph can be

partitioned into at most k cliques. Mujuni and Rosamond [57] investigated this problem from the point of view of parameterized complexity. They showed that choosing the number of cliques as a parameter makes this problem fixed-parameter tractable. More precisely, if k is the number of cliques, then a kernel bounded by k^2 can be obtained in polynomial time. Formally we can define the problem as follows. *Given an undirected connected graph $G = (V,E)$ and an integer k, where $n = |V|$ is the number of vertices. Is there a set of at most k cliques in G such that each edge in E has both its endpoints in exactly one of the selected cliques?* The main result of their work showed that clique partition has a kernel bounded by k^2; hence it is fixed-parameter tractable.

Cygan et al. [19] have shown a doubly exponential lower bound for Edge Clique Cover parameterized by the number of cliques, obtaining tight complexity bounds for this problem. They presented a polynomial-time algorithm that reduces an arbitrary instance of `3-CNF-SAT` with n variables and m clauses to an equivalent ECC instance (G,k) with $k = O(\log n)$ and $|V(G)| = O(n+m)$. According to the authors, Edge Clique Cover is the first example of a natural parameterized problem for which doubly exponential upper and lower bounds were proved.

The significant advances made in the area of fixed-parameter tractability of many computationally intractable problems can provide valuable insights to design algorithms that provably solve these problems optimally as long as the size of the parameter is small. However, coping with big-data instances requires algorithmic solutions that are scalable and amenable to parallelization. The classical characterization of "efficient algorithm" as one that runs in polynomial time on the input size may no longer be entirely satisfactory. To be practically viable, it is essential that the algorithms run in nearly linear or sub-linear time on the input size.

In spite of the significant practical implications of the ECC problem, there has been only a handful of works concerned with the design and implementation of efficient algorithms that meets real-world requirements on performance and reliability.

Kellerman [42] proposed a heuristic algorithm for determining keyword conflicts, which is related to the edge clique cover problem. Unfortunately, that algorithm does not guarantee that the number of cliques is maximal. Then Kou et al. [47] improved Kellerman's heuristic and used that algorithm to solve the edge clique cover problem.

Gramm et al. [27] proposed a heuristic algorithm and showed how to improve the performance of Kellerman's [42] heuristic algorithm from $O(m^2n)$ to $O(mn)$ while preserving the same result they got for their exact algorithm, where m is the number of the edges and n is the number of vertices of the given graph.

The goal of the edge clique cover problem is to minimize the number of overall cliques. A less well-studied but equally important goal is to minimize the number of individual assignments of vertices to cliques. This latter problem is also NP-hard and known as "Assignment Minimum Clique Cover". In some works, this problem is addressed as the "Local Clique Cover". Ennis et al. [23] proposed algorithm for assignment-minimum clique coverings. They experimented with their post-processing algorithm with both exact and heuristic algorithms.

In a recent approach, Conte et al. [17] have introduced $O(m\Delta)$ heuristic, due to Bron et al. [7], to cover all edges of a given graph; where m is the number of edges, and Δ is the highest degree of the graph. Their heuristic processes uncovered edges one after another to find a clique cover. They presented experimental data for large sparse networks.

2.4 Conclusion

Analyzing characteristics in networked data, such as graphs, can yield important information on the modeled structure. Efficient representation of network data is critical to addressing algorithmic challenges in analyzing massive data sets using graph-theoretic abstractions. Graph decomposition is a way to discover essential features in graphs and compactly represent sparse graphs. We see tremendous interest in analyzing graphs by solving graph problems from the literature, such as graph coloring problems, clique cover prob-

lems, and counting and enumerating local topological structures, such as triangles. This thesis focuses on edge clique cover problems, assignment-minimum edge clique cover, triangle counting & enumeration, k-count, and triangle centrality. The following chapters will discuss our algorithms and comparative results with the state-of-the-art algorithms.

Chapter 3

Covering Large Complex Networks by Cliques using Ordered vertices

3.1 Introduction

In diverse areas as sparse matrix determination and complex network analysis, two kernel operations are to identify and compute with dense or otherwise highly connected subgraphs [43, 36]. Identifying special interest groups or characterizing information propagation in social networks are examples of frequently performed operations [82]. Efficient representation of network data is critical to addressing algorithmic challenges in the analysis of massive data sets using graph theoretic abstractions. In this chapter, we propose sparse-matrix data structures to enable the compact representation of graph data and use an existing sparse matrix framework [32] to design efficient algorithms for the Edge Clique Cover (ECC) problem.

The literature shows the ECC problem's applications in disparate areas from the extensive theoretical investigation on the ECC problem and its variants. Hossain et al. [35] describe a branch-and-bound approach to determine sparse Jacobian matrices. Given the sparsity pattern of the Jacobian, the problem is to find a partition of the columns into structurally orthogonal column groups of smallest cardinality. The intersection graph associated with the sparse Jacobian is obtained as a collection of cliques. The coloring algorithm exploits this clique decomposition to guide branching steps and cutting down on extraneous computation. In computational biology, the study of the protein complex identification problem is to identify overlapping protein complexes in protein-protein interaction net-

works [6]. Modelling this problem as a graph problem aims to decompose the network into a small collection of cliques. In sensory science, a frequently occurring task is concerned with the concise representation of pairwise interaction of products with many attributes [22]. One can give this pairwise information in a tabular form called "compact letter display". Minimizing redundant information is the main challenge of this problem. Ennis and Ennis [22] showed that this problem can be posed as a variant of the ECC problem. Heuristics have been proposed in the literature to approximately solve ECC problem while there are only a few exact methods which are usually limited to solving small instance sizes.

A recent approach is described by Gramm et al. in [27], where they introduce and analyze data reduction techniques to shrink the instance size without sacrificing the optimal solution. The main idea is that with small enough instance sizes, exact algorithms may become feasible.

In this chapter, we present a compact representation of network data based on sparse matrix data structures [35]. Employing this data structure, we propose an algorithm to solve the edge clique cover problem. Our implemented algorithm is based on an algorithm due to Kellerman [42]. In the next chapter, we propose another method to solve this exact problem (edge clique cover), which shows a significant improvement over state-of-the-art (algorithm by Conte et al. [17]).

Our approach is based on the observation presented by Hasan et al. [32]. In that work, the authors show that for a sparse matrix $X \in \mathbb{R}^{m \times n}$, the row intersection graph of X is isomorphic to the adjacency graph of $XX^\top$. Similarly, the column intersection graph of A is isomorphic to the adjacency graph of $X^\top X$. From this observation, we see, the subset of columns corresponding to nonzero entries in row i induces a clique in the adjacency graph of $X^\top X$. Analogously, the subset of rows corresponding to nonzero entries in column j induces a clique in the adjacency graph of $XX^\top$. We followed this approach in our work because even if matrix X is sparse, matrices $X^\top X$ and $XX^\top$ are most likely dense. We exploit the connection between sparse matrices and graphs in the reverse direction. We

can define a sparse matrix, *intersection matrix*, for a given graph (or network) such that graph algorithms of interest can be expressed in terms of the associated intersection matrix. We use the existing sparse matrix computational framework to solve graph problems for this structural reduction [32]. In the literature, we also see that the graph algorithms can be expressed in the language of linear algebra by exploiting this duality between graphs and sparse matrices [43, 44]. Graph kernel operations such as breadth-first search, graph connected components etc. can be expressed as sparse matrix-matrix multiplication [43]. This mapping between graphs and sparse matrices can be exploited in data analysis and algorithm design. This is especially important because of the unstructured nature of data access and the memory-bound nature of graph algorithms. Sparse matrix infrastructure that is available in very high-level programming languages such as MATLAB and Python and the efficient linear algebraic software libraries implementing BLAS/sparse BLAS provide a realistic and practical approach to a wide range of graph algorithms.

3.1.1 Organization of the Chapter

We organize the rest of the chapter in the following way. In Section 3.2, we describe the classical data structure of graphs: adjacency representation, and adjacency list, followed by the intersection matrix representation, enabling an efficient representation of pairwise information. Section 3.3 describes our vertex-centric method to compute an edge-clique cover for a given graph. The central idea of our method is inspired by an algorithm due to Kellerman [42]. For ease of presentation, we discuss the algorithm in graph-theoretic terms. Results from numerical experiments on a standard collection of test instances are provided in Section 3.4. Fianally, Section 3.5 concludes the chapter.

3.2 Representation of Sparse Graphs

Adjacency representation is known as the classical data structure for graphs. However, adjacency representations such as adjacency matrix and adjacency list are inadequate for

efficient computer implementation of many graph operations. Using the adjacency matrix for representing a sparse graph is costly. On the other hand, typical adjacency list implementations employ pointers where indirect access leads to poor cache utilization. For an undirected graph implementation, the adjacency list typically stores each edge twice. This redundancy is avoided where the edges incident on each vertex is stored in the sorted order of vertex labels [80].

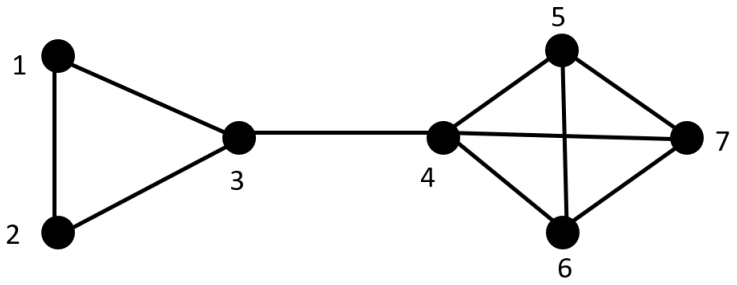


Figure 3.1: Example of an undirected graph

Figure 3.1 presents an undirected graph (G) with seven vertices and ten edges and Figure 3.2 shows the adjacency representation of G.

Each row and column of the adjacency matrix (Figure 3.2-a) represents an adjacency, i.e. neighborhood, of a vertex. Every non-zero entry indicates an edge, thus $A(i, j) = 1$ identifies an edge between vertices i and j. The adjacency matrix also stores zero if there is no edge between two vertices. Therefore, the total memory requirement is $O(n^2)$, where $n =$

	1	2	3	4	5	6	7
1		x	x				
2	x		x				
3	x	x		x			
4			x		x	x	x
5				x		x	x
6				x	x		x
7				x	x	x	

(a) Adjacency matrix

1	→	2	3		
2	→	1	3		
3	→	1	2	4	
4	→	3	5	6	7
5	→	4	6	7	
6	→	4	5	7	
7	→	4	5	6	

(b) Adjacency list

Figure 3.2: Adjacency representation of graph shown in Figure 3.1

	1	2	3	4	5	6	7
1	x	x					
2	x		x				
3		x	x				
4			x	x			
5				x	x		
6				x		x	
7				x			x
8					x	x	
9					x		x
10						x	x

Figure 3.3: Intersection representation of graph shown in Figure 3.1

$|V|$. On the other hand, in the adjacency list representation (Figure 3.2-b), we store ordered pairs and their permutations, e.g. (i, j) and (j, i) and need to spend additional memory space to store the address of the next pointer (not shown in the figure for brevity).

The close connection between the set intersection representation and undirected graphs has a long been known in graph-theory literature.

A collection of subsets $\{S_1, S_2, ..., S_n\}$ where S_i, $i = 1, 2, \ldots, n \subset U$ for a universal set U can be represented by a simple undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. To see this, we associate vertex $i \in V$ with set S_i and for $i \neq j$, $\{i, j\} \in |E|$ whenever $S_i \cap S_j$ is not empty. On the other hand, Szpilrajn-Marczewski [79] showed that given an undirected simple graph $G = (V, E)$, there is an universal set U and a collection of of subsets $S_1, S_2, \ldots, S_n$ there is a bijection between vertices in V and the collection of subsets $S_1, S_2, \ldots, S_n$ such that $\{i, j\} \in E$ if and only if $i \neq j$, $S_i \cap S_j \neq \emptyset$.

We now introduce the intersection matrix representation, enabling an efficient representation of pairwise information. The trivial intersection matrix is only the simplest matrix that belongs to the class of intersection matrix. The transpose of the trivial intersection matrix is also known as the incidence matrix [43].

Intersection Matrix: An *intersection representation* of graph G is a matrix $X \in \{0, 1\}^{k \times n}$, where $k \leq m$, in which for each column j of X there is a vertex $v_j \in V$ and $\{v_i, v_j\} \in E$ whenever there is a row l for which $X(l, i) = 1$ and $X(l, j) = 1$.

For $k = m$, the rows of X represent the edge list sorted by vertex labels. Therefore, matrix X can be viewed as an assignment to each vertex a subset of m labels such that there is an edge between vertices i and j if and only if the inner product of the columns i and j is 1. Since the input graph is unweighted, the edges are simply ordered pairs, and can be sorted in $O(m)$ time. Unlike the adjacency matrix which is unique (up to a fixed labelling of the vertices) for graph G, there can be more than one *intersection matrix* representation associated with graph G [1]. We exploit this flexibility to store a graph in a structured and space-efficient form.

Figure 3.3 shows an intersection matrix representation of graph shown in Figure 3.1. Each column represents a vertex, and each row represents an edge between vertices of corresponding columns.

Graph algorithms can be effectively expressed in terms of linear algebra operations [43]. We now show a connection between a graph and its sparse matrix representation and cast the edge clique cover problem using linear algebra operations.

Column Intersection Graph: The *column intersection graph* associated with matrix $X \in \mathbb{R}^{m \times n}$ is a graph $G = (V, E)$ in which for each column k of X there is a vertex $v_k \in V$ and $\{v_i, v_j\} \in E$ whenever there is a row l for which $X(l,i) \neq 0$ and $X(l,j) \neq 0$.

Theorem 3.1. *Let $X \in \{0,1\}^{m \times n}$ be the intersection matrix associated with a graph $G = (V,E)$, and consider $B = X^\top X$. Then the adjacency graph of matrix B is isomorphic to graph G.*

Proof. Consider an arbitrary edge $e_k = \{v_i, v_j\}$ of graph G. By construction, row k of the intersection matrix X has $X(k,i) = X(k,j) = 1$ and $X(k,l) = 0$ for $l \notin \{i, j\}$. Since there are no multiple edges in G, there is one and only one such row k corresponding to edge e_k. Element $B(i,j)$ is the inner product of column vectors i and j of matrix X. The inner product is 1 if and only if $X(k,i) = X(k,j) = 1$. Thus, e_k is in E if and only if $B(i,j) = 1$ implying that it is an edge connecting vertices v_i and v_j of the adjacency graph of matrix B. This proves the theorem. □

X=

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

(a) Intersection matrix

B= $X^\top X$ =

$$\begin{bmatrix} 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 3 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 3 \end{bmatrix}$$

(b) Adjacency matrix

Figure 3.4: Finding adjacency matrix from an intersection matrix

Theorem 3.1 establishes the desired connection between a graph and its sparse matrix representation (see Figure 3.4). For a vertex $v \in V$ we define by $N_v = \{w \in V \mid \{v,w\} \in E\}$ the set of its neighbors. The *degree* of a vertex v, denoted $d(v)$, is the cardinality of set N_v. The following result follows directly from Theorem 3.1.

Corollary 3.2. *The diagonal entry $B(i,i)$ where $B = X^\top X$ and X is the intersection matrix of graph G, is the degree $d(v_i)$ of vertex $v_i \in V, i = 1,\ldots,n$ of graph $G = (V,E)$.*

We can easily get the adjacency matrix from the intersection matrix X using $B = X^\top X$. For example, consider the undirected graph (G) shown in Figure 3.1. Figure 3.4-a shows an intersection matrix representation of the given graph. Using matrix-matrix multiplication ($X^\top X$) we get the adjacency matrix (Figure 3.4-b), which is relatively dense.

Intersection matrix X defined above represents an edge clique cover of cardinality m for graph G. Each edge $\{v_i, v_j\}$ constitutes a clique of size 2 (e.g. Figure 3.3). In the intersection matrix X, the clique (edge) is represented by row k with $X(k,i) = X(k,j) = 1$ and other entries in the row being zero. In general, column indices l in row k where $X(k,l) = 1$ constitutes a clique on vertices v_l of graph G. Thus the edge clique cover problem can be cast as a matrix compression problem.

minECC Matrix Problem. Given $X \in \{0,1\}^{m \times n}$ determine $X' \in \{0,1\}^{k \times n}$ with k minimized such that the intersection graphs of X and X' are isomorphic.

	1	2	3	4	5	6	7
1	x	x	x				
2			x	x			
3				x	x	x	x

Figure 3.5: Intersection representation of the edge clique cover of the graph shown in Figure 3.1

Figure 3.5 shows how we can represent the edge clique cover of a given graph using the intersection matrix representation. Here each row represents a clique, and columns represent corresponding vertices.

Storing a graph G in a sparse intersection matrix form requires less space than the adjacency representation. If we store the graph as a clique cover, it requires even less space than the sparse intersection matrix representation. With the help of a real network, we can show how an intersection matrix representation can save lots of space. For a concrete example, we consider the `LiveJournal` Graph collected from the Stanford Network Analysis Platform (SNAP) data sets [20]. `LiveJournal` is a free online community with almost 10 million members. The graph we have used from SNAP consists of 4,847,571 vertices (members) and 68,993,773 edges (the connection between members). To store this graph in the adjacency and sparse intersection representation would require the following space.

- Adjacency matrix stores n^2 integers, where n is the number of vertices. The size of an integer is 4 bytes, and therefore, the total space required to store the graph is **87540 gigabytes**.

- Adjacency list requires $(n+2m)$ nodes to store the graph, where n is the number of vertices and m is the number of edges. Each node needs 16 bytes to store an integer and the address of the next node. Therefore, the total space required to store the graph is **2.13 gigabytes**. However, accessing data from the adjacency list is costly.

- Sparse intersection matrix requires to store $2(m+n)$ integers. The total space required is **0.55 gigabytes**.

Our computer implementation to solve the edge clique cover problem uses a sparse matrix framework of DSJM [32], and all computations are expressed in terms of intersection matrices. Let us discuss the data structures used to store the sparse intersection matrices in DSJM.

Compressed Sparse Row (CSR) Data Structure

We can implement CSR using three arrays: *rowptr*, *colind* and *value*. In *colind* array we store the column index of each non-zero element and value of that element is stored in *value* array. However, we do not require the array, *value*, to solve the edge clique cover problem beacuse we asume the nonzero value is 1. The *rowptr* indices the array, *colind*. *colind* and *rowptr* are integer arrays. The size of *colind* is $2m$ and the size of *rowptr* is $m+1$, where m indicates the number of rows of the matrix, which is the number of edges of the graph. $rowptr(i)$ is the column index of first non-zero entry of row i. Suppose $rowptr(1)$ is 1. So its column index is $colind(rowptr(1))$. We can also access all non-zero entries of a row using CSR data structure. If $rowptr(1)$ is 1 and $rowptr(2)$ is 3 then we have $rowptr(2) - rowptr(1) = 3 - 1 = 2$ non-zero entries in row 1. We get the column indices and values of the non-zero entries of row 1 form *colind* and *value* arrays. We can access elements of row i as $colind(rowptr(i))$ to $colind(rowptr(i+1) - 1)$.

Compressed Sparse Column (CSC) Data Structure

The CSC data structure is similar to the CSR. However, the three arrays are *colptr*, *rowind* and *value*. Again, we do not need array, *value*, for solving the edge clique cover problem as we assume the nonzero value is 1. *colptr* indices the array *rowind*. $colptr(j)$ is the row index of the first nonzero element of column j. *rowind* and *colptr* are integer arrays. The size of *rowind* is $2m$ and the size of *colptr* is $n+1$, where n indicates the number of columns of the matrix. We can access elements of col j as $rowind(colptr(j))$ to $rowind(colptr(j+1) - 1)$.

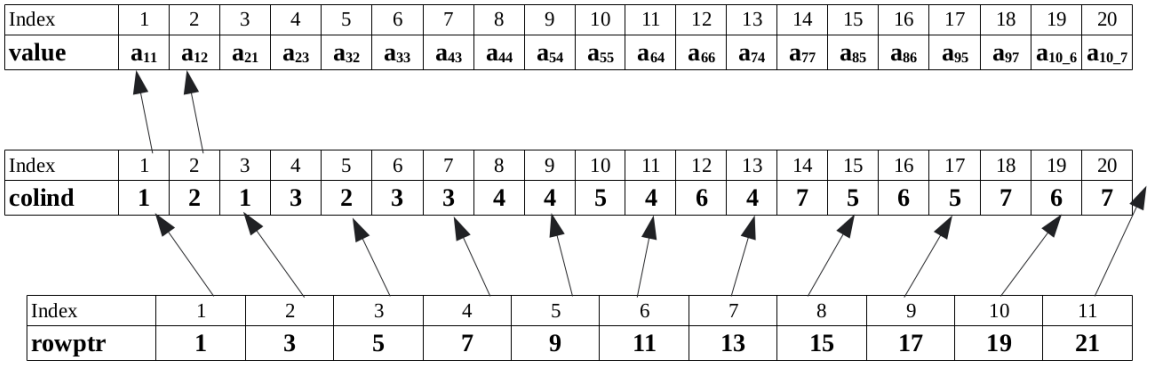


Figure 3.6: CSR data structure of intersection matrix shown in Figure 3.3

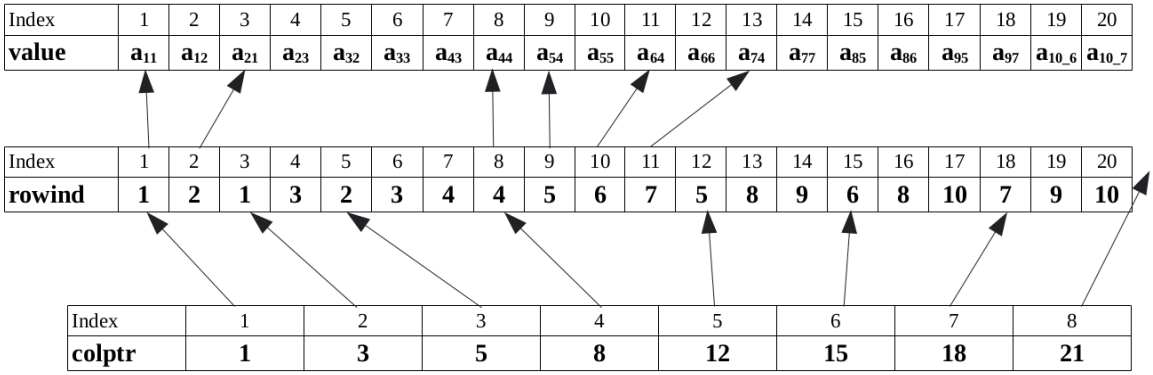


Figure 3.7: CSC data structure of intersection matrix shown in Figure 3.3

An Example How CSR and CSC are Used to Store Sparse Intersection Matrix

We can efficiently perform graph operations using CSR and CSC without explicitly constructing the graph. Using CSC, we get the non-zero entry of any specific column, and from their row index and then using CSR, we can find if any non-zero entries in that row are the neighbors of that column.

A simple example will make it easier to understand. Below, the non-zero entries of matrix X in Figure 3.3 is given then *colind*, *rowptr* arrays of CSR and *rowind*, *colptr* arrays of CSC are given in Figure 3.6 and in Figure 3.7 respectively.

3.3 A Heuristic for Clique Cover

We are now ready to describe our vertex-centric method to compute an edge-clique cover for a given graph. The central ideas of our method is inspired by an algorithm due to

Kellerman [42]. For ease of presentation we discuss the algorithm in graph theoretic terms. However, our computer implementation uses sparse matrix framework of DSJM [32] and all computations are expressed in terms of intersection matrices.

There is a close connection between the vertex clique cover of a graph $G = (V, E)$ and the coloring of vertices of the complement graph $\overline{G} = (V, \overline{E})$ where $\overline{E} = \left\{ \{u, v\} \mid \{u, v\} \notin E \right\}$ [30]. A graph has a vertex clique cover of size k iff its complement graph can be colored with k colors such that adjacent vertices have different colors. In the classical graph coloring problem, vertices of the graph are partitioned into subsets (colors) such that pairs of vertices connected by an edge are in different subsets. The optimization version asks for the partition with the smallest number of subsets. It is well-known that the greedy coloring heuristic is sensitive to the order in which the vertices are processed (see [35]). Consider an optimal coloring of graph G and order the vertices in the nondecreasing color index. It is not difficult to see that the greedy heuristic on graph G with the given order of the vertices produces optimal coloring. We experimentally verify that the ECC heuristic is sensitive to the ordering in which the vertices are processed. We employ three vertex ordering algorithms from the literature: Largest-degree order (LDO), Degeneracy order (DGO), and Incidence-degree Order (IDO) prior to applying the heuristic [42]. We recall that $d(v) = |N_v|$ denotes the degree of vertex v in graph $G = (V, E)$.

- **(LDO)** Order the vertices such that $\{d(v_i), i = 1, \ldots, n\}$ is nonincreasing.
- **(DGO)** Let $V' \subseteq V$ be a subset of vertices of G. The subgraph induced by V' is denoted by $G[V']$. In this thesis, one of the methods we use to order the vertices is using the degeneracy. Assume the vertices $V' = \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ have already been ordered. The i^{th} vertex in DGO is an unordered vertex u such that $d(u)$ is minimum in $G[V \setminus V']$ where, $G[V \setminus V']$ is the graph obtained from G by removing the vertices of set V' from V.

  Let us demonstrate the DGO ordering for the graph shown in Figure 3.1. Table 3.1 shows step by step calculation of the degeneracy to order the vertices. In each step, we

Table 3.1: Steps of DGO

Step	*Column removed*	*Minimum degree*	*DGO ordering*
1	1	2	{1}
2	2	1	{2,1}
3	3	1	{3,2,1}
4	7	3	{7,3,2,1}
5	6	2	{6,7,3,2,1}
6	5	1	{5,6,7,3,2,1}
7	4	0	{4,5,6,7,3,2,1}

choose the column which has the minimum degree in the induced subgraph $G[V \setminus V']$ and remove that column. Therefore, after all the steps, we get the ordering of vertices $\{4,5,6,7,3,2,1\}$.

- **(IDO)** Assume that the first $k-1$ vertices $\{v_1 \ldots, v_{k-1}\}$ in incidence-degree order have been determined. Choose v_k from among the unordered vertices that has maximum degree in the subgraph induced by

$$\{v_1, \ldots, v_k\}$$

3.3.1 Algorithm

Now, we present the algorithm for the ECC problem. Let the vertices of graph $G = (V,E)$ be ordered in one of DGO, LDO, and IDO: $v_1, \ldots, v_n$. Also, let $V_{\mathcal{P}} = \{v_1, \ldots, v_{i-1}\}$ denote the vertices that have been assigned to one or more cliques $\{C_1, \ldots, C_{k-1}\}$ and v_i be the vertex currently being processed. Denote by set

$$W = \{v_j \mid j < i \text{ and } \{v_i, v_j\} \in E\}$$

the neighbors of v_i in $V_{\mathcal{P}}$. An edge $\{u,v\} \in E$ is said to be *covered* if both of its incident vertices have been included in some clique; otherwise the edge is *uncovered*. The task is

to assign v_i to one or more of the existing cliques (or create a new clique) such that each edge incident on v_i that connects to a vertex in $V_{\mathcal{P}}$ is covered by a clique. There are three possibilities:

Case I. W is empty: Create a new clique $C_k = \{v_i\}$

Case II. W is not empty:

Case a. There is a clique $C_l, l \in \{1,\ldots,k-1\}$ such that $W = C_l$: add v_i to C_l

Case b. There is no such clique:

i. If $C_l \subset W$ for some l, add v_i to C_l and update W by $W \setminus C_l$.

ii. If there are uncovered edges after step II(b(i)), create a new clique from an existing clique and add v_i and the incident edges until W is empty.

The complete algorithm is presented in Algorithm 1.

We argue that the cliques $C_1, C_2, ..., C_k$ returned by the algorithm constitutes an edge clique cover for the input graph G.

The main **for**-loop (line 2) reads the next vertex (i) from the ordered list of vertices and tries to include it in one of the existing cliques, or creates new clique(s) with vertex i included. If vertex i has no neighbor (i.e. $W = \emptyset$) in $V_{\mathcal{P}}$, a new clique gets created (line 6). If the neighbor set W is not empty, the algorithm tries to identify existing cliques C_l that are subsets of W and assigns vertex i to each of them (lines 9 – 15, **Case 2. a.** and **Case 2. b. i.**). This step covers edges of the form $\{i, i'\}$ where $i' \in C_l$, $C_l \subset W$. Finally, the **while**-loop (line 16) covers the remaining edges (**Case II. b. ii.**) of the form $\{i, i'\}$ where $i' \in S, S = W \cap C'_l$, $l' \in \{1, 2, \ldots, l\}$ with $|S|$ maximum. The maximality on $|S|$ ensures that each newly created clique covers largest number of uncovered edges. For a graph $G = (V, E)$ each edge is a clique of size 2 so that set E constitute an (trivial) ECC. Therefore, each edge of input graph G eventually gets assigned to one of the cliques output by Algorithm 1.

The above discussion can be summarized in the following result.

Algorithm 1 VertexOrderedECC (W, $list$)

```
1  k ← 0                                              ▷ Number of cliques
2  for index = 1 to N do                              ▷ N denotes the number of vertices
3      i ← list[index]                                ▷ list contains the vertices in a predefined order
4      if W = ∅ then                                  ▷ W ← {j | j < i and {i, j} ∈ E}
5          k ← k + 1
6          C_k ← {i}                                  ▷ C_k denotes k^th clique
7      else
8          U ← ∅                                      ▷ Contains neighbors of i, which are in the cliques
9          for l = 1 to k do
10             if C_l ⊆ W then
11                 C_l ← C_l ∪ {i}
12                 U ← U ∪ C_l
13                 if U = W then
14                     break
15             W ← W \ U
16         while W ≠ ∅ do
17             Max ← ∅
18             MIN_l ← 0
19             for l = 1 to k do
20                 if |Max| < |(C_l ∩ W)| then
21                     Max ← (C_l ∩ W)
22                     MIN_l ← l
23             l ← MIN_l
24             k ← k + 1
25             C_k ← (C_l ∩ W) ∪ {i}
26             W ← W \ C_l
27 return C_1, C_2, ..., C_k
```

Lemma 3.3. *The collection* $\{C_1, C_2, \ldots, C_k\}$ *computed by Algorithm 1 constitutes an ECC of graph G.*

3.4 Numerical Results

In this section, we provide results from numerical experiments on selected test instances. The graph instances are chosen from standard benchmark collections that are used in the literature for ECC and closely related graph problems such as, graph coloring and graph partitioning. The data set for the experiments is obtained from the University of Florida Sparse Matrix Collection [20]. Instances **chesapeake, delaunay_n10 to 13, as-22july06** are from "10th DIMACS Implementation Challenge" benchmark collection for graph clustering and graph partitioning. Instances **ca-GrQc, as-735, Wiki-Vote, p2p-Gnutella04, Oregon-1** are from "Stanford Network Analysis Platform (SNAP)" collection. These instances represent social networks from variety of apllications. We also consider the data set for Compact Letter Displays used by Gramm et al. [29]. The experiments were performed using a PC with 3.4G Hz Intel Xeon CPU, 8 GB RAM running Linux. The implementation language was *C*++ and the code was compiled using $-O2$ optimization flag with a *g*++ version 4.4.7 compiler.

A short description of the data set for our experiments is as follows:

- **chesapeake:** Symmetric, undirected graph and contains 39 vertices and 170 edges.
- **delaunay_n10 to 13:** The graphs are symmetric and undirected. The minimum degree is 3 for all of them and the maximum degrees are 12, 13, 14 and 12 respectively.
- **as-22july06:** The graph is symmetric and undirected having maximum degree 2.4K and minimum degree 1.
- **ca-GrQc:** General Relativity and Quantum Cosmology network covers scientific collaboration between authors in this field. This graph contains an undirected edge from i to j, if author i co-authored a paper with author j.

- **as-735:** An autonomous system which represents a communication network of who-talks-to whom.
- **Wiki-Vote:** This data set contains voting data of Wikipedia till January 2008 where the contest was between volunteers to become one of the administrator. There is a directed edge from node i to node j if user i voted for user j.
- **p2p-Gnutella04:** A snapshoot of Gnutella peer-to-peer file sharing network on August 04, 2002. A directed graph where nodes represent hosts and edges represent connection between hosts.
- **Oregon-1:** Undirected graph where autonomous system peering information is inferred from Oregon route-views on May 26, 2001.
- **Triticale, winter wheat and oilseed rape yield trials:** These instances are from the application "compact letter display" [29] to test ECC algorithms.

Gramm et al. [30] presented their heuristic method inspired by Kellerman [42] and Kou et al. [47] to find edge clique cover and implemented their method in `Objective Caml 3.08.3`. We will refer thier method as `Gramm-Method`. Our "Vertex-centric edge clique cover" algorithm, `VO-ECC`, is also inspired by Kellerman's algorithm [42]. The implementation is in *C++* and gives better results than `Gramm-Method`. In the latter chapter, we propose another method to solve the edge clique cover problem, "Edge-centric edge clique cover" algorithm, `EO-ECC`, and compare the results with the state-of-the-art algorithm by Conte et al. [17]. Conte et al. (referred to as `Conte-Method`) presented their method for finding edge clique cover and showed that their algorithm performs better than other edge clique cover algorithms.

Here, in Table 3.2, we present comparative results between `Gramm-Method`, `VO-ECC`, `Conte-Method`, and `EO-ECC` reporting clique cover sizes and running time for five real-world instances. n represents the number of vertices and m represents the number of edges

of the graph. $|C|$ represents number of cliques required to cover all the edges, and t represents the running time in seconds.

Table 3.2: Test Results for Real-World Matrices

Graph			Gramm-Method		Conte-Method		VO-ECC		EO-ECC	
Name	m	n	$\|C\|$	t (sec)	$\|C\|$	t (sec)	$\|C\|$	t (sec)	$\|C\|$	t (sec)
triticale1	55	13	**4**	0	**4**	0.1	**4**	0	**4**	0
triticale2	86	17	6	0	**5**	0.03	**5**	0	**5**	0
wheat1	4847	124	50	0.15	50	0.08	50	0.06	**49**	0.01
wheat2	4706	121	**48**	0.13	50	0.07	**48**	0.05	**48**	0.01
wheat3	3559	97	34	0.03	34	0.06	32	0.01	**31**	0.01

Now we present test results of `VO-ECC`. Test results for the selected test instances from group DIMACS10 and SNAP are reported in Table 3.3 and Table 3.4 respectively. Test results for Compact Letter Display are reported in Table 3.5.

Table 3.3: Test Results for DIMACS10 Matrices

Graph			Natural	DGO	LDO	IDO
Name	m	n	$\|C\|$	$\|C\|$	$\|C\|$	$\|C\|$
chesapeake	170	39	90	**79**	83	80
delaunay_n10	3056	1024	1300	**1223**	1302	1268
delaunay_n11	6127	2048	2610	**2482**	2617	2527
delaunay_n12	12264	4096	5228	**4973**	5264	5061
delaunay_n13	24547	8192	10489	**9937**	10541	10121
as-22july06	48436	22963	34695	34772	**34568**	34666

Table 3.4: Test Results for SNAP Matrices

Graph			Natural	DGO	LDO	IDO
Name	m	n	$\|C\|$	$\|C\|$	$\|C\|$	$\|C\|$
ca-GrQc	14496	5242	3791	3879	**3777**	3900
as-735	13895	7716	9055	9108	**8985**	9038
Wiki-Vote	103689	7115	43497	45530	**42482**	45491
p2p-Gnutella04	39994	10876	38475	**38474**	38475	**38474**
Oregon-1	23409	11174	15736	15807	**15631**	15857

Table 3.5: Test Results for Real-World (Compact Letter Displays) Matrices [29]

Graph			VO-ECC	Insert-Absorb	Clique-Growing	Search Tree
Name	m	n	$\|C\|$	$\|C\|$	$\|C\|$	$\|C\|$
Triticale 1	55	13	4	4	4	4
Triticale 2	86	17	5	5	5	5
Wheat 1	4847	124	50	56	50	49
Wheat 2	4706	121	48	50	48	48
Wheat 3	3559	97	32	39	32	31
Rapeseed 1	576	47	20	20	20	20
Rapeseed 2	1040	57	20	20	20	20
Rapeseed 3	1260	64	24	24	24	24
Rapeseed 4	1085	62	19	19	19	19
Rapeseed 5	1456	64	19	19	19	19
Rapeseed 6	1416	70	27	27	27	27
Rapeseed 7	1758	74	26	29	27	25
Rapeseed 8	1128	59	17	17	17	17
Rapeseed 9	1835	76	30	30	30	30

For comparison we also show the ECC results where no specific vertex ordering is employed, in addition to ordering algorithms LDO, DGO, and IDO. Column labelled "Natural" reports the ECC result when the vertices are processed in the order they are specified in the data file. On DIMACS10 instances, degeneracy order (DGO) gives the best result except for instance named `as-22july06`. On SNAP instances largest-degree order (LDO) is the overall winner. Note that on both sets of test instances ordered approach produces strictly better ECC compared with *Natural*. We remark that *OCaml* implementation by Gramm et al. [28], fails (hangs) to run on DIMACS10 and SNAP instances. As such no comparison of the ECC quality (size) can be made. Table 3.5 displays results using our degree ordered method and two other algorithms discussed by Gramm et al. [29]. *Insert Absorb* and *Search Tree* require exponential running time while *Clique Growing* method is an improved implementation of the heuristic of Kellerman [42]. *Search Tree* is an exact method that produces optimal ECC. *VO-ECC* reports the best edge clique cover of our implementation. It is evident from the table that our method produces optimal or near optimal

(off by 1) ECC.

3.5 Conclusion

In this chapter, we have shown that the connection between large networks and their sparse matrix representation can be exploited to employ efficient techniques to find edge clique covers by employing graph-sparse matrix duality as in sparse matrix determination literature [37, 38]. The edge clique cover problem is recast as a sparse matrix determination problem. The notion of *intersection matrix* provides a unified framework that facilitates compact representation of graph data and efficient implementation of graph algorithms. The adjacency matrix representation of a graph can potentially have many nonzero entries since it is the product of an intersection matrix with its transpose. We have shown that, similar to graph vertex coloring problem, the ECC problem is sensitive to ordering of the vertices.

Chapter 4

Covering Large Complex Networks by Cliques using Ordered Edges

4.1 Introduction

In Chapter 3, we presented a "vertex-centric" algorithm for solving the edge clique cover (`VO-ECC`) problem where the central idea of our method is inspired by an algorithm due to Kellerman [42]. There we proposed an efficient representation of network data and showed that the algorithm is sensitive to the ordering in which the vertices are processed. In this chapter, we use a compact representation of network data based on sparse matrix data structures [32] and present an "edge-centric" minECC method motivated by the works of Bron et al. [7], and E. Tomita et al. [81] for finding clique covers. Moreover, we explore a variant of edge clique cover, namely, "Assignment Minimum Clique Cover" and develop a scalable algorithm to heuristically solve the problem.

We experimentally verify that our "edge-centric" minECC algorithm is sensitive to the ordering in which the edges are processed. Therefore, we also present three edge ordering techniques and use them in our algorithm. We will refer to our "edge-centric" minECC algorithm as `EO-ECC` in the following sections.

In a recent approach, Conte et al. [17] have introduced $O(m\Delta)$ algorithm to cover all edges of a given graph, where m is the number of edges, and Δ is the highest degree of the graph. For the rest of this chapter, we will refer to Conte's algorithm as `Conte-Method`.

Finally, we provide results from numerical experiments on selected large test instances. From the literature, we see that most of the experiments show results for small graphs

[68, 23, 28, 27, 29, 30]. However, Conte et al. [17] presented their experimental results for large graphs, but their algorithm is unable to handle large real-life instances (for instance, *com-LiveJournal*). On the other hand, our algorithm finds assignment minimum edge clique cover for graphs with more than eighty million edges. Moreover, our `EO-ECC` algorithm has performed significantly better compared with `Conte-Method` in running time and scalability on all tested instances.

4.1.1 Organization of the Chapter

The chapter is organized as follows. Section 4.2 presents a motivating example for solving the edge clique cover problem focusing on the assignment minimum edge clique cover. In Section 4.3, we present the new edge-centric minECC algorithm. An essential ingredient of our algorithm is to select edges incident on the vertex being processed in specific orders. The details of the implementation steps are described, followed by the presentation of the ECC algorithm. This section contains a detailed discussion of the computational complexity of the algorithm.

Section 4.4 contains results from elaborate numerical experiments. We choose different network data sets consisting of real-world networks and synthetic instances.

In Section 4.5, we describe the parallel implementation of our "edge-centric" edge clique cover algorithm, `EO-ECC` for finding edge clique cover for a given graph.

Finally, the chapter is concluded in Section 4.6.

4.2 A Motivating Example

We used *intersection matrix* (discussed in Section 3.2), to store the given undirected graph. Still, it requires a row for each edge. This section presents an example showing that a clique cover can store all the graph information using less space.

Consider the undirected graph (G) in Figure 4.1 having $|V| = 9$ and $|E| = 18$. In a trivial clique cover, we can consider an edge as a clique. Using the intersection matrix, we can

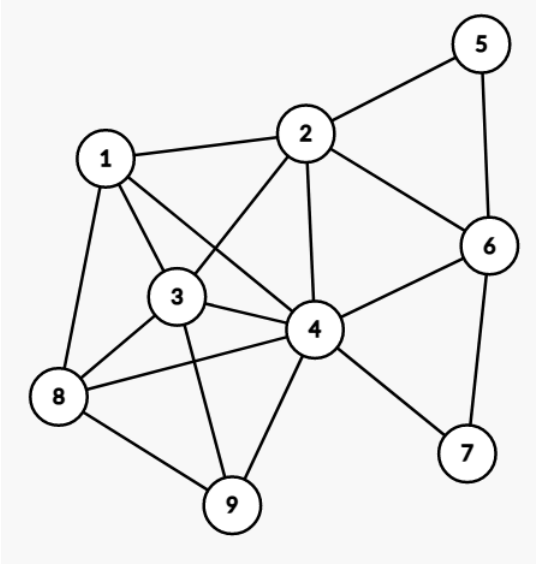


Figure 4.1: An undirected graph G

store the clique cover with $2 \times 18 = 36$ nonzeros. Let, using an algorithm, we get an edge clique cover for the given graph G (as shown in Figure 4.2).

For this edge clique cover, we observe the number of nonzeros to represent this cover decreases to 21 (See Figure 4.3).

Still, more optimization is possible, and we can achieve that with some post-processing steps. One important post-processing step is eliminating redundant cliques from the edge clique cover. By the word "redundant clique" we mean a clique where other cliques already cover all of it's edges. For example, Clique-4 in Figure 4.2 is a redundant clique. Because edges $\{2,6\}$, $\{2,4\}$ and $\{4,6\}$ are covered by Clique-6, Clique-2 and Clique-5 respectively. So, after this post-processing step, we get five cliques, and the total number of nonzeros to represent this processed cover decreases to 18 (see Figure 4.4).

Another important post-processing step minimizes the number of nonzeros to store the cover, creating an assignment minimum edge clique cover. The goal of the assignment-minimum-cover is to minimize the number of individual assignments of vertices (number of nonzeros) to cliques. In the minECC problem, there might be cases where more than one cliques cover the same edges, and therefore, the clique cover is not assignment minimum clique cover. However, we apply this post-processing step after we find the minimum edge

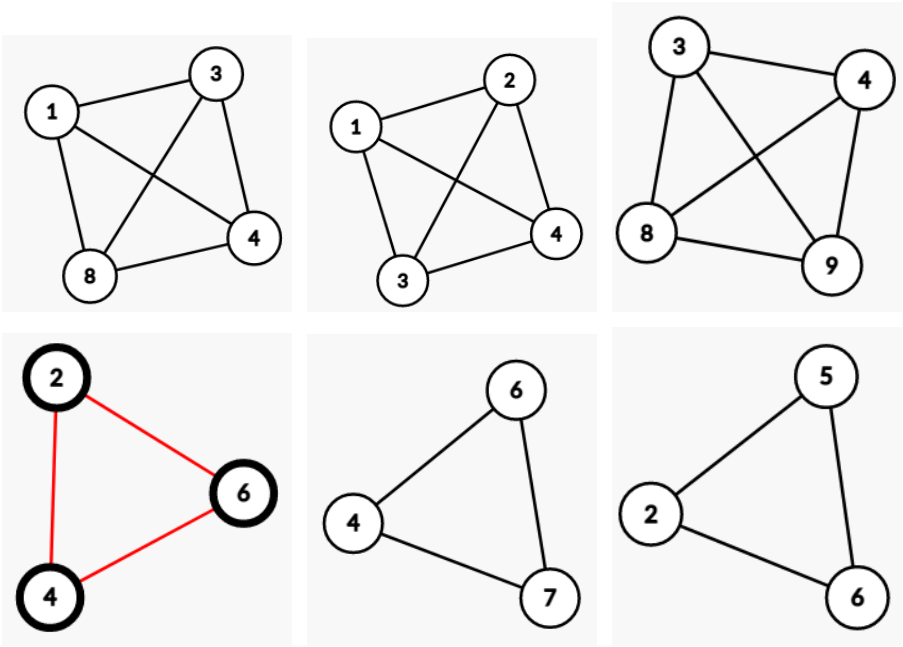


Figure 4.2: An edge clique cover for the graph shown in Figure 4.1. Let, we label the cliques from left to right and from top to bottom. Therefore, the top-left clique is Clique-1 and the bottom-right clique is Clique-6

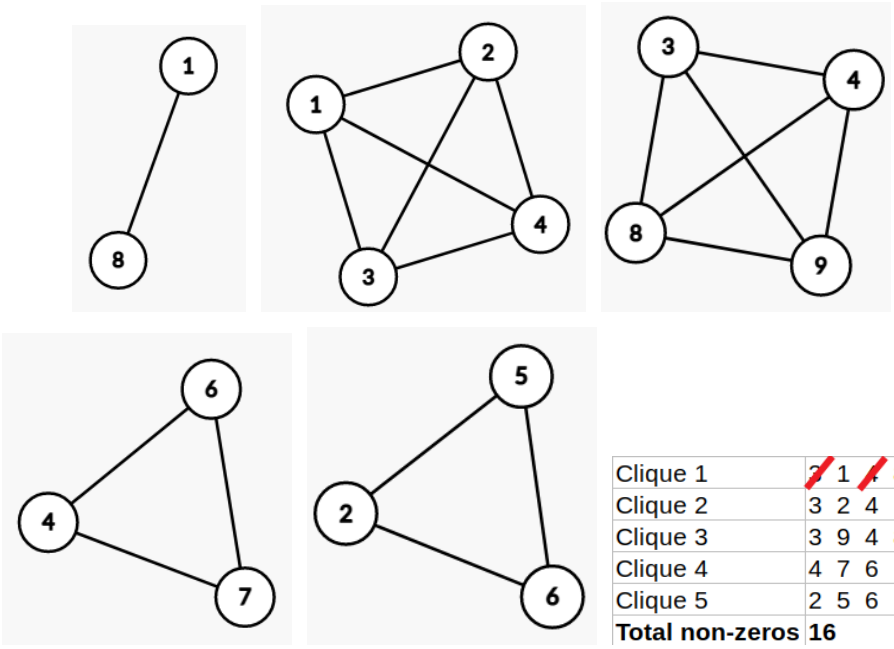
Clique 1	3 1 4 8
Clique 2	3 2 4 1
Clique 3	3 9 4 8
Clique 4	4 6 2
Clique 5	4 7 6
Clique 6	2 5 6
Total non-zeros	**21**

Figure 4.3: Total nonzeros required to store ECC shown in Figure 4.2

clique cover. Because this post-processing step does not decrease the number of cliques but tries to remove vertices (nonzeros) from the cliques of the cover. For example, Clique-1 of the ECC, presented in Figure 4.2, contains six edges. Here, we can remove vertices 3 and 4 because all the edges incident to these vertices are covered in other cliques (Clique-2 and Clique-3). Therefore, the total number of nonzeros to represent this processed cover decreases to 16 (Figure 4.5).

Clique 1	3 1 4 8
Clique 2	3 2 4 1
Clique 3	3 9 4 8
~~Clique 4~~	~~4 6 2~~
Clique 5	4 7 6
Clique 6	2 5 6
Total non-zeros	**18**

Figure 4.4: Total nonzeros required to store processed ECC after removing redundant cliques



Clique 1	3 1 4 8
Clique 2	3 2 4 1
Clique 3	3 9 4 8
Clique 4	4 7 6
Clique 5	2 5 6
Total non-zeros	**16**

Figure 4.5: Processed ECC and number of nonzeros

4.3 An Edge-Centric minECC Algorithm

We have implemented our algorithm based on algorithms due to C. Bron et al. [7], and E. Tomita et al. [81]. For ease of presentation, we discuss the algorithm in graph-theoretic terms. However, our computer implementation uses a sparse matrix framework of DSJM [32] and expresses all computations in terms of intersection matrices.

We introduce edge ordering techniques and process edges according to that order. In the following section, we will discuss the edge ordering techniques followed by our improved algorithm and post-processing steps.

Definition 4.1. *Intersect(L1,L2):* We are given two sorted lists: $L1$, and $L2$. Now we

can find the set intersection, S, between these two lists, by merging $L1$ and $L2$. The time complexity is given by $O(max\{|L1|,|L2|\})$.

Definition 4.2. *FindNeighbors(v):* For an undirected graph G, we can define the neighbor set of a vertex $v \in V$ as $Neighbor(v) = \{w \mid \{v,w\} \in E\}$

We store the given graph using the column intersection matrix (see Section 3.2). In this column intersection matrix X, the number of rows is $|E| = m$. Each row contains only two nonzeros: v, and another is the neighbor of v. Because a row corresponds to an edge and stores only two endpoints of that edge. Let, $d(v)$ be the degree of vertex v. Now, while finding the neighbor of vertex v, we only look up the rows where v is present. Therefore, we do a constant time (two times) operation in each row. For a vertex v, we lookup $|d(v)|$ number of rows, therefore, for all vertices, total lookup is $\sum_{v=1}^{n} |d(v)|$, which is equals to $2 \times m$. Therefore, to find the neighbor set of all the vertices, our algorithm takes linear time, $O(m)$.

Definition 4.3. *FindCommonNeighbors(Neighbor(v_1),Neighbor(v_2)):* We are given lists of neighbors for vertex v_1 and vertex v_2: $Neighbor(v_1)$, and $Neighbor(v_2)$ respectively. We use *FindNeighbors(v)*, to get these lists. Now using *Intersect(L1,L2)* we can get the common neighbor of vertices v_1 and v_2. The running time of this method for using in our algorithm (`EO-ECC`) will be discussed later in the following section.

4.3.1 Edge Ordering Techniques

Vertex Ordering

We described vertex ordering techniques in the previous chapter. We recall that $d(v)$ denotes the degree of vertex v in graph $G = (V,E)$. Let *Vertex_Order* be a list of vertices of graph G using one of the ordering schemes below.

- **Largest-Degree Order (LDO) (see [35]):** Order the vertices such that $\{d(v_i), i = 1,\ldots,n\}$ is nonincreasing.

- **Degeneracy Order (DGO) (see [25, 71]):** Let $V' \subseteq V$ be a subset of vertices of G. The subgraph induced by V' is denoted by $G[V']$. Assume the vertices $V' = \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ have already been ordered. The i^{th} vertex in DGO is an unordered vertex u such that $d(u)$ is minimum in $G[V \setminus V']$ where, $G[V \setminus V']$ is the graph obtained from G by removing the vertices of set V' from V.

- **Incidence-Degree Order (IDO) (see [16]):** Assume that the first $k-1$ vertices $\{v_1 \ldots, v_{k-1}\}$ in IDO have been determined. Choose v_k from among the unordered vertices that has maximum degree in the subgraph induced by $\{v_1, \ldots, v_k\}$.

Edge Ordering

After the vertices have been ordered using one of the above schemes, the algorithm proceeds to choose a vertex in that specific order, which has at least one uncovered incident edge. If there is more than one uncovered edge incident on the vertex being processed, the order in which the edges are processed (i.e., to include in a clique) is as follows. Place all the edges $\{u,v\}$ before $\{p,q\}$ in an ordered edge list, *Edge_Order*, such that vertex u or vertex v is ordered before vertices p and q in *Vertex_Order* list.

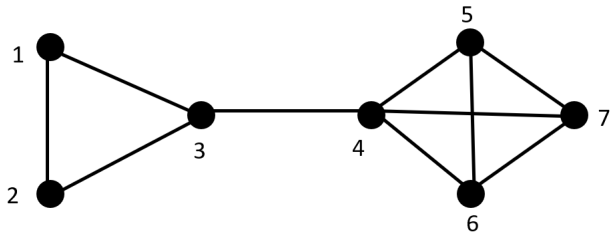


Figure 4.6: An example of an undirected graph

Figure 4.6 shows an undirected graph. $\{4,3,5,6,7,1,2\}$ would be a list with LDO. Then the edge list induced by the *Vertex_Order* will have the following form.

$$Edge\_Order = \{\{4,3\},\{4,5\},\{4,6\},\{4,7\},\{3,1\},\{3,2\},\{5,6\},\{5,7\},\{6,7\},\{1,2\}\}$$

Edge Selection

We select an edge to $\{u,v\} \in E$ to include in a new clique if $\{u,v\}$ is uncovered and ordered before all uncovered edges in $Edge\_Order$. The clique that gets constructed with edge $\{u,v\}$ may cover other uncovered edges that are further down the list.

We consider three variants of edge selection for our algorithm, denoted by L, D, and I.

- **L**: In this variant, the set of vertices are ordered using the Largest-Degree Ordering (LDO) scheme. We select a vertex u in that order and then return all the uncovered edges of the form $\{u,v\}$.
- **D**: All the vertices are ordered using Degeneracy Ordering (DGO) scheme. Select a vertex u in that order, and then return all the uncovered edges of the form $\{u,v\}$.
- **I**: Finally, this variant orders the set of vertices using the Incidence-Degree Ordering (IDO) scheme. We select a vertex u in that order and return all the uncovered edges $\{u,v\}$.

4.3.2 Algorithm

Let $E_{\mathcal{P}} = \{e_1, \ldots, e_{i-1}\}$ denote the edges that have been assigned to one or more cliques $\{C_1, \ldots, C_{k-1}\}$ and $e_i = \{v_i, v_j\}$ be the edge currently being processed. The set

$$W(v_i, v_j) = \big\{ v_l \mid \{v_i, v_l\}, \{v_j, v_l\} \in E \big\}$$

denotes the set of common neighbors of two vertices v_i and v_j. The task is to assign edge $\{v_i, v_j\}$ (if not covered yet) to one new clique and add its common neighbors if they satisfy clique properties.

For an uncovered edge $\{v_i, v_j\}$, the algorithm creates a new clique $C_k = \{v_i, v_j\}$. Then there are two cases.

Case I. $W(C_k)$ is empty: do nothing.

Case II. $W(C_k)$ is not empty:

i. Take a vertex v_l from $W(C_k)$ such that v_l is ordered before all the vertices $v_p \in W(C_k)$, $l \neq p$.

ii. Update $W(C_k)$ by $W(C_k) \cap Neighbor(v_l)$

iii. Include v_l in C_k.

iv. Repeat steps i to iv until $W(C_k)$ is empty.

The complete algorithm is presented below.

Algorithm 2 EO-ECC ($Edge\_Order$)

```
Input: Edge_Order, set of edges in a predefined order using schemes L, D, or I
1  k ← 0                                              ▷ Number of cliques
2  for index = 1 to m do                              ▷ m is number of edges
3      {u, v} ← Edge_Order[index]
4      if {u, v} is not covered then
5          W ← FindCommonNeighbors(u, v)
6          if W = ∅ then
7              k++
8              C_k ← {u, v}
9              Mark {u, v} as covered.
10         else
11             k++
12             C_k ← {u, v}
13             Mark {u, v} as covered.
14             while W ≠ ∅ do
15                 let t be a vertex in W
16                 W ← W \ t
17                 if {t, s} ∈ E for each s ∈ C_k then
18                     Mark {t, s} as covered
19                     C_k ← C_k ∪ {t}
20                     FindCommonNeighbors(W, FindNeighbors(t))
21 return C_1, C_2, ..., C_k
```

Lemma 4.4. *Let $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$ be the clique decomposition produced by* `EO-ECC` *algorithm. Then, for indices i not equal to j for $C_i, C_j \in \mathbb{C}$, it holds that C_i is not a subset of C_j.*

Proof. Let $C_1, C_2, \ldots, C_k$ be the cliques of the edge clique cover in their order of discovery by `EO-ECC`, namely, C_i is discovered before C_j iff $i < j$. We will prove that:

- no clique C_i is contained in another clique C_j, where $i \neq j$

Suppose by contradiction that $C_i \subseteq C_j$ for $i \neq j$. First, we observe that it cannot be that $C_i = C_j$ as they must be found from one uncovered edge, and the first discovered of the two would cover all the edges of the other. Thus, it must be $C_i \subset C_j$. Second, it must be $i < j$ for the same reason: if it were $j < i$, then C_i would contain only covered edges.

Now we get a contradiction for using the method to find the set of common neighbors. This method finds the common neighbor list of all the vertices of C_i. It includes such vertex in C_i, and updates the common neighbor list. Therefore, our algorithm does not leave any edge which is incident to all the vertices of C_i. So, C_j will start creating the clique with a covered edge and therefore will create a clique with all covered edges. But, this is not possible according to our construction of edge clique cover (i.e. picking an uncovered edge for creating a new clique). □

Corollary 4.5. *Each $C_i, i = 1, 2, \ldots, k$ is a maximal clique in $G = (V, E)$.*

Proof. Let $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$ be the clique decomposition produced by `EO-ECC` algorithm. Now if C_i is not a maximal clique than we shall get another clique C_j in the given decomposition where C_i is the subset of C_j. It is not possible according to Lemma 4.4. □

Lemma 4.6. *For each edge $\{i, j\} \in E$ there is at least one clique C_l such that $i, j \in C_l$.*

Proof. Algorithm 2 terminates when there is no edge to process. Therefore, all the edges are covered by at least one clique. □

Lemma 4.7. *Algorithm 2 is correct.*

Proof. Algorithm 2 is correct because it gives a maximal clique cover (Corollary 4.5) and all the edges are covered by at least one clique (Lemma 4.6). □

Remark 4.8. Let $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$ be the clique decomposition produced by `EO-ECC` algorithm. Here for any graph, $k \leq m$, where m is the number of edges. Because, in the trivial case, we can consider an edge as a clique, therefore $k = m$. When we cover more than one edge in a clique, we get $k < m$. There is no chance where k can be larger than m.

Lemma 4.9. *The operation to check whether all edges of the given graph are covered or not takes $O(m)$ time in total.*

Proof. In the column intersection matrix, each column represents a vertex. To get the index of an edge, we intersect two columns of that matrix. If there is an edge between these two vertices, they intersect in exactly one row and using that index; we keep track of whether it is covered.

The total intersection cost is $\sum_{i=1}^{n} \frac{\rho_i \times (\rho_i - 1)}{2} = O(m)$, where ρ_i is the degree of vertex v_i.

Note, we check each edge twice because we select the edges from its two incident vertices. Still, the running time is $O(m)$ for checking the status of all the edges. □

Lemma 4.10. *The operation* FindNeighbors *in `EO-ECC` to calculate neighbor set of vertex $v \in V$ is implemented in linear space, taking $O(m)$ time to find neighbor sets for all the vertices.*

Proof. In `EO-ECC`, we need to know the set of neighbors of at most two vertices at a time. We have two n size arrays, where n is the number of vertices. Therefore, the space requirement is linear.

Recall, we store the given graph using the column intersection matrix (see Section 3.2). In column intersection matrix X, the number of rows is $|E| = m$. Each row contains only two nonzeros: v, and another is the neighbor of v. Let, $d(v)$ is the degree of vertex v. Now, while finding the neighbor of vertex v, `EO-ECC` only checks the rows that contains v. Therefore, we do a constant time (two times) operation in each row. For a vertex v, we check $|d(v)|$ number of rows, therefore, for all vertices, we check $\sum_{v=1}^{n} |d(v)|$, which is equals to $2 \times m$. Therefore, to find neighbor set of all the vertices, our algorithm takes linear time, $O(m)$. □

Lemma 4.11. *The operation* FindCommonNeighbors *in* `EO-ECC` *is implemented in linear space and to find all the cliques C_i of ECC, it takes $O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$ time, where k is the number of cliques and ρ_i is the size of clique C_i.*

Proof. In `EO-ECC`, at a time, we need to compute one common neighbor set between two vertices. At this time, we do not need to know about the common neighbor set of other vertices. Therefore, we have one n size array, where n is the number of vertices to store the common neighbors set. So, the space requirement is linear.

According to our algorithm, we start with an uncovered edge $\{u,v\}$ and include these vertices in a new clique C_i. Then the *FindCommonNeighbors* operation takes $O(max\{d(u), d(v)\})$ time, where $d(v)$ denotes the degree of vertex v, to find the neighbor set between these two vertices. Now the while loop at line 14 of Algorithm 2 runs until the common neighbor set W is empty and each time it includes a vertex from W to the current clique C_i. Therefore, if ρ_i denotes the size of C_i, then we say, this loop runs for ρ_i times. Now, inside the *FindCommonNeighbors* operation, it merges two sorted lists (lists of neighbors) where the set of vertices that this operation returns after each call has at least one fewer vertices and in each iteration, this merging cost decreases at least by one. So, to find clique C_i, total cost would be $\rho_i + (\rho_i - 1) + \cdots + 2 + 1$. Therefore cost of this operation to find a clique C_i, is $(\frac{\rho_i \times (\rho_i - 1)}{2})$. As we have k cliques, the total cost is $O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$. □

Remark 4.12. The time required for the *FindCommonNeighbors* operation, presented in Lemma 4.11, depends on the number of times the `EO-ECC` covers an edge.

Let, for an undirected graph G, we get a clique cover with k number of cliques using the `EO-ECC` algorithm.

1. If the graph is triangle-free ($k = m$), the running time of the *FindCommonNeighbors* operation is $O(m)$.

2. If G is a clique of size n ($k = 1$), the running time of the *FindCommonNeighbors* operation is $O(m)$.

3. If G is a graph with $n-l$ cliques and each clique contains exactly $l+1$ vertices, then the running time of the *FindCommonNeighbors* operation is $(n-l)\frac{(l+1)\times l}{2} > m$.

Here, cases 1 and 2 are the two extreme cases for any given graphs, and case 3 is an example of a graph where the operation *FindCommonNeighbors* has its worst-case scenario.

From Lemma 4.11, we know that the operation *FindCommonNeighbors* in `EO-ECC`, takes

$$O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$$

time, where ρ_i is the size of clique C_i.

For Case 1, $\rho_i = 2$ for $i = 1,\ldots,k$, and $k = m$. Therefore solving the equation we get, $O(m)$.

Similarly for Case 2, $\rho_i = n$ for $i = 1,\ldots,k$, and $k = 1$. We know, for a complete graph, the number of edges, $m = \frac{n(n-1)}{2}$. Therefore, we get total running time $O(m)$.

Case 3 is an example of a graph where we have $n-l$ cliques of the same size, $l+1$, and all cliques have a common sub-clique, l-clique. Therefore, $\frac{l\times(l-1)}{2}$ number of edges are covered $n-l$ times. For this case, $\rho_i = l+1$ for $i = 1,\ldots,k$, and $k = (n-l)$. According to Lemma 4.11, the total time for this operation is $(n-l)\frac{(l+1)\times l}{2}$.

From the discussion on these cases, it is clear that the runtime for the operation *FindCommonNeighbors* depends on the number of times the ECC covers an edge.

Theorem 4.13. *For an undirected and connected graph $G = (V,E)$, where $|V| = n$, and $|E| = m$, Algorithm 2 takes $O(n+m)$ space and $O(m + \sum_{i=1}^{k} \frac{\rho_i\times(\rho_i-1)}{2})$ time to find edge clique cover $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$, where ρ_i denotes the size of clique C_i and $(k \leq m)$.*

Proof. `EO-ECC` stores the given graph in an intersection matrix, where rows represent an edge and columns represent vertices. Each row consists of two vertices, and therefore a total of $O(2m)$ space is required, where m is the number of edges. We also calculate the degree of each vertex to get their order list. Therefore, we require $O(n)$ space to store that degree and order information of the vertices. `EO-ECC` algorithm has two primary operations:

FindNeighbors, and *FindCommonNeighbors*. According to Lemma 4.10 and 4.11, both operations use $O(m)$ space. Therefore, total space required for the Algorithm 2 is linear, i.e. $O(n+m)$.

`EO-ECC` requires an ordered list of vertices. We use an existing sparse matrix framework [32] to design the efficient algorithm, `EO-ECC`. Hossain et al. [32] showed a graph coloring algorithm along with a vertex ordering scheme and proved that accessing this sparse matrix framework takes linear time, $O(m)$ for ordering the vertices. To check whether all edges of the given graph are covered or not, `EO-ECC` algorithm takes $O(m)$ time (Lemma 4.9). Then Lemma 4.10 shows the operation *FindNeighbors* takes $O(m)$ time to calculate the list of neighbors for all the vertices. The operation *FindCommonNeighbors* depends on the number of times the ECC covers an edge (Remark 4.12) and takes $O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$ time to get the edge clique cover $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$ (4.11), where ρ_i is the size of clique C_i. Therefore, the total time requires for the `EO-ECC` algorithm is $O(m + \sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$. □

4.3.3 Removing Redundant Cliques

The example in Section 4.2 shows we have two post-processing methods after we get the edge clique cover using Algorithm 2. The first post-processing method is to remove the redundant cliques. A clique is said to be a redundant clique if other cliques cover all the edges of that clique. Kou et al. [47] proposed a similar concept to removing redundant cliques after having the edge clique cover using Kellerman's [42] algorithm. However, those approaches work well for small graphs, and we cannot test that algorithm for large graphs. We present our post-processing method in Algorithm 3.

Algorithm 3 RemoveRedundantCliques ($\mathbb{C} = C_1, C_2, \ldots, C_k$)

```
1 k' ← k                          ▷ k' is the number of cliques after removing redundant cliques
2 for i = 1 to k do                                              ▷ k is number of cliques
3     if all the edges {u, v} ∈ C_i is covered more than once then
4         Remove clique C_i
5         k' ← k' − 1
6 return C_1, C_2, ..., C_k'
```

Lemma 4.14. *The post-processing step, Algorithm 3 takes* $O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$ *time, where k is the number of cliques and* ρ_i *is the size of clique* C_i.

Proof. Algorithm 2 finds an edge clique cover, $\mathbb{C} = C_1, C_2, \ldots, C_k$. Then Algorithm 3 takes that clique cover as an input and checks cliques one by one. For a clique C_i, algorithm checks all the edges of that clique, whether those edges are covered more than once or not. Therefore, it takes $O(\frac{\rho_i \times (\rho_i - 1)}{2})$ time, where ρ_i is the size of clique C_i. To check all the cliques (k number of cliques), this algorithm takes $O(\sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$ time. □

4.3.4 Assignment Minimum Edge Clique Cover

The second post-processing method is for achieving assignment minimum edge clique cover. We pass the given graph through Algorithm 2 and Algorithm 3 and then this post-processing method takes that edge clique cover without the redundant cliques as input. After we pass this post-processing step, the number of cliques remains the same in the edge clique cover, but the size of the cliques may decrease, and therefore we get assignment minimum edge clique cover. Ennis et al. [23] presented an algorithm for assignment minimum edge clique cover. However, their backtracking algorithm becomes costly for large graphs, especially when they have many maximal cliques. In the following section, we have presented comparative results between `EO-ECC` and Ennis's algorithm.

For our post-processing method we get an edge clique cover without redundant cliques, $\mathbb{C}' = C_1, C_2, ..., C_{k'}$. Now we examine all the cliques one after another. For a clique C_i, we consider a subgraph, S_v that contains all the edges incident to vertex $v \in C_i$. Now if all the edges of this subgraph S_v, is covered by other cliques $C_j \in \mathbb{C}'$, and $i \neq j$, then we can remove this vertex v from C_i. We presented the complete method in Algorithm 4.

Lemma 4.15. *The post-processing step, Algorithm 4, takes* $O(\sum_{i=1}^{k'} \rho_i + \frac{\rho_i \times (\rho_i - 1)}{2})$ *time, where* k' *is the number of cliques and* ρ_i *is the size of clique* C_i.

Proof. After we remove all the redundant cliques using Algorithm 3 we get an edge clique cover, $\mathbb{C}' = C_1, C_2, \ldots, C'_k$. Then Algorithm 4 takes that clique cover as input and checks

Algorithm 4 AM-ECC ($\mathbb{C}' = C_1, C_2, \ldots, C_{k'}$)

```
1  for i = 1 to k' do
2      for u ∈ C_i do
3          flag_u ← 0
4      for edge {u, v} in C_i do
5          if edge {u, v} ∈ C_i is covered only once then
6              flag_u ← 1
7              flag_v ← 1
8      for u ∈ C_i do
9          if flag_u == 0 then
10             Remove u from C_i
```

cliques one by one. For each clique C_i, this algorithm has the following steps: (1) set the flags of each vertex of C_i to zero. (2) examine each edge, whether those are covered exactly once or not. If an edge is covered exactly once, it means, only C_i covers that edge, and therefore we cannot remove the vertices incident to that edge. Set the flag of such vertices to one. (3) Examine the flags of each vertex, and remove the vertex $u \in C_i$ if its flag value is still zero.

Step 1 takes ρ_i times for clique C_i. Then Step 2 takes $\frac{\rho_i \times (\rho_i - 1)}{2}$ time as it examine all the edges of clique C_i. Finally, Step 3 takes ρ_i time. Therefore total time required to examine all the cliques is $O(\sum_{i=1}^{k'} \rho_i + \frac{\rho_i \times (\rho_i - 1)}{2})$. □

4.4 Numerical Results

In this section, we provide results from numerical experiments on selected real-life and generated test instances.

10th Discrete Mathematics and Theoretical Computer Science (DIMACS10) data sets and Stanford Network Analysis Platform (SNAP) data sets for the experiments are obtained from the University of Florida Sparse Matrix Collection [20]. We obtain some real-world instances, such as triticale, winter wheat, and rapeseed trials instances from [29], where they presented the application of "compact letter display" to test Edge Clique Cover (ECC) algorithms. We also test our algorithms for some large social networks, available in the

Network Repository [69]. Here, the most extensive graph we tested had 35 million edges and 4 million vertices. We then generated 182 Erdős-Rényi and Small-World instances using a general-purpose, high-performance system for graph and network manipulation and analysis, **Stanford Network Analysis Platform** (`SNAP`) [50]. The number of edges of these generated graphs is varied from 800 edges to 72 million edges.

The experiments were mainly performed using a PC with 3.4GHz Intel Xeon CPU, with 8 GB RAM running Linux. The implementation language was *C++* and the code was compiled using $-O2$ optimization flag with a *g++* version 4.4.7 compiler. We employed the High-Performance Computing system (Graham cluster) at Compute Canada for large instances that could not be handled by the PC.

Conte et al. [17] presented their method to find edge clique cover and showed that their algorithm performs better than other edge clique cover algorithms. Therefore, we compared our result for the `EO-ECC` algorithm with Conte's method. Conte et al. implemented their method in *Java* language, and we ran their code in the same environment we used for our algorithm. In this chapter, we show that our algorithm takes less time than Conte's method to find an edge clique cover. For 219 test instances (from DIMACS10, SNAP, Real-World, Small-World, and Erdős-Rényi groups), where the number of edges varies between 170 and 7.6×10^7, our `EO-ECC` algorithm produces smaller or equal size clique covers than the `Conte-Method` [17].

With post-processing, we achieve assignment minimum edge clique cover. Ennis et al. [23] presented a post-processing method to find assignment minimum edge clique cover. They tested their method after they got edge clique cover using different existing algorithms. Therefore, we compare our assignment minimum edge clique cover result with Ennis's result. We show that our algorithm finds assignment minimum edge clique cover. However, Ennis et al. could not test their algorithm for large graphs. Nevertheless, in our result, we present results for large graphs too.

This section refers to Conte's method as `Conte-Method`, `Ennis-ALG` will refer to the

Table 4.1: Test Results (number of cliques) for DIMACS10 Graphs

Graph			Number of cliques			
Name	m	n	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
chesapeake	170	39	**75**	76	**75**	76
delaunay_n10	3056	1024	1250	**1233**	1275	1241
delaunay_n11	6127	2048	2485	**2449**	2544	2481
delaunay_n12	12264	4096	4993	**4906**	5095	4939
delaunay_n13	24547	8192	9989	**9881**	10211	9920
delaunay_n14	49122	16384	19974	**19672**	20435	19855
delaunay_n15	98274	32768	39923	**39501**	40876	39782
delaunay_n16	196575	65536	79933	**78792**	81528	79445
delaunay_n17	393176	131072	159900	**157792**	163321	158851
delaunay_n18	786396	262144	319776	**315684**	326741	317987
com-DBLP	1049866	317080	238854	237713	**237685**	**237685**
belgium_osm	1549970	1441295	1545183	1545183	1545183	1545183
delaunay_n19	1572823	524288	639349	**631354**	653383	635877
delaunay_n20	3145686	1048576	1279101	**1262843**	1307080	1271229
delaunay_n21	6291408	2097152	2557828	**2525301**	2613106	2542333

Ennis's algorithm, and our algorithm as `EO-ECC`. Our algorithm (`EO-ECC`) employed three different techniques to order the edges described in the previous section. `EO-ECC` has three variants associated with the three different edge ordering schemes `D`, `L`, and `I`. They are: `EO-ECC-D`, `EO-ECC-L`, and `EO-ECC-I` respectively.

4.4.1 Analyzing Clique Size Distribution

Number of Cliques

Test results for the selected test instances from group *DIMACS10* are reported in Table 4.1. Here, n represents the number of vertices and m represents the number of edges of the graph. In this table, the smallest cardinality clique cover is marked in **bold**.

For comparison, we show the results of `Conte-Method`, `EO-ECC-D`, `EO-ECC-L`, and `EO-ECC-I`. For twelve out of fifteen instances, `EO-ECC-D` gives the least number of cliques to cover all the edges of a given graph. For one graph, `EO-ECC-L` and `EO-ECC-I` produce smaller cardinality ECC. Finally, all algorithms give the same result for two out of fifteen instances. Figure 4.7 shows the improvement (percentage) in clique cover size using

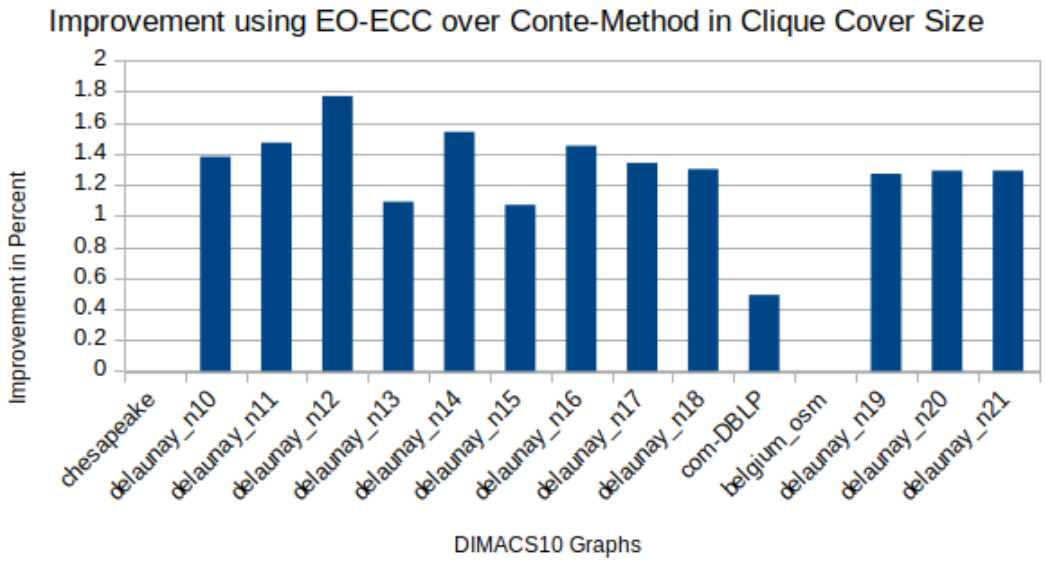


Figure 4.7: Improvement in clique cover size using EO-ECC

`EO-ECC` over `Conte-Method`. We report the best results from all the twenty versions of Conte's method. For all the DIMACS10 instances, our `EO-ECC` finds an equal or smaller size clique cover than `Conte-Method`.

Besides DIMACS10 selected instances, we compare these algorithms on 182 generated instances where the graph's number of edges is varied from 800 to 7.2×10^7. Using SNAP tool [50], we generated 72 "Small-world" and 110 "Erdős-Rényi" graphs. For all the instances, `EO-ECC` produces smaller (47.3%) or equal (52.7%) cardinality of the clique cover than `Conte-Method`. The detailed test results of these instances are reported in Table 1 of Appendix *A*.

Maximal Clique Size Dimension

To find a maximal size clique in the clique cover, `EO-ECC-L` and `EO-ECC-I` are more effective. These algorithms gave maximal size cliques for all of the instances: `SNAP`, `DIMACS10`, `Erdȍs-Rényi`, and `Small-World`. For example, `Conte-Method` finds equal (40% cases) or smaller (60% cases) sized cliques in the cover than the one found by `EO-ECC` for the `DIMACS10` instances.

Average Clique Size

From a clique cover, we compute the maximal average size of the cliques using $\frac{\sum_{i=1}^{k}|C_i|}{|C|}$, where C is a clique cover, and k is the size of the clique cover. In all the cases `EO-ECC` achieves the maximal average size than `Conte-Method`. In 27.55% cases, `Conte-Method` gives an equal average size of the cliques, and for the rest of the cases, it gives a smaller average size of cliques than `EO-ECC`.

Trivial Maximal Clique

`EO-ECC` gives maximum size cliques in all cases, and it also achieves the maximum average size of cliques. Therefore, it provides less number of trivial maximal cliques (i.e., a clique of size 2). In 72% of the cases, `EO-ECC` gives less trivial maximal cliques than `Conte-Method`, and on average, this value is 12.7% smaller than that of `Conte-Method`. In the rest of the cases, both algorithms give an equal number of trivial cliques.

Relative Difference in Number of Cliques

In this chapter, we have presented two post-processing methods: to remove redundant clique (Algorithm 3), and to achieve assignment minimum edge clique cover (Algorithm 4). Therefore, after the post-processing, we can say we do not have any redundant cliques. So, we would get a more trivial maximal clique, i.e., a clique of size two. Again, we want to compare the results between `EO-ECC` and `Conte-Method` for the number of cliques, but we ignore the clique of size two. Mathematically we can represent this as follows. $|\mathbb{C}''| = |\mathbb{C}'| - |\mathbb{T}|$, where $\mathbb{C}'$ is the edge clique cover we get from `EO-ECC` and using two post-processing algorithms, $\mathbb{T} \subseteq \mathbb{C}'$, where $\mathbb{T}$ contains the cliques of size two, and $\mathbb{C}''$ is the set of cliques where $\mathbb{C}'' \subseteq \mathbb{C}'$ and it contains all the cliques of size more than two.

Table 4.2: Relative Difference in Number of Cliques between Conte-Method and EO-ECC

Graph				$\lvert ECC\rvert - \lvert trivial\_cliques\rvert$		Relative
				Conte-Method	EO-ECC	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100\ \%$
DIMACS10	chesapeake	170	39	47	47	0.00
	delaunay_n10	3056	1024	1050	984	6.29
	delaunay_n11	6127	2048	2125	1989	6.40
	delaunay_n12	12264	4096	4251	3982	6.33
	delaunay_n13	24547	8192	8587	7966	7.23
	delaunay_n14	49122	16384	17098	15922	6.88
	delaunay_n15	98274	32768	34141	31807	6.84
	delaunay_n16	196575	65536	68208	63647	6.69
	delaunay_n17	393176	131072	136504	127277	6.76
	delaunay_n18	786396	262144	273228	254478	6.86
	com-DBLP	1049866	317080	153392	152094	0.85
	belgium_osm	1549970	1441295	2417	2414	0.12
	delaunay_n19	1572823	524288	546594	508896	6.90
	delaunay_n20	3145686	1048576	1092346	1017901	6.82
	delaunay_n21	6291408	2097152	2185359	2036084	6.83
SNAP	as-735	12572	7716	2279	2279	0.00
	ca-GrQc	14484	5242	1990	1954	1.81
	ca-HepTh	25973	9877	5100	5038	1.22
	p2p-Gnutella04	39994	10879	861	845	1.86
	p2p-Gnutella24	65369	26518	912	897	1.64
	p2p-Gnutella25	54705	22687	754	742	1.59

Table 4.2 – *Continued on next page*

Table 4.2 – *Continued from previous page*

Graph				$\lvert ECC\rvert - \lvert trivial\_cliques\rvert$		Relative
				Conte-Method	EO-ECC	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100\ \%$
	p2p-Gnutella30	88328	36682	1430	1403	1.89
Real-World	Rapeseed 6	1416	70	26	25	3.85
	Rapeseed 7	1758	74	23	22	4.35
Erdős-Rényi	er-n6	8.00E+06	1.00E+06	731	731	0.00
	er-2n6	1.60E+07	2.00E+06	696	695	0.14
	er-3n6	2.40E+07	3.00E+06	683	683	0.00
	er-4n6	3.20E+07	4.00E+06	634	634	0.00
	er-5n6	4.00E+07	5.00E+06	660	659	0.15
	er-6n6	4.80E+07	6.00E+06	712	712	0.00
	er-7n6	5.60E+07	7.00E+06	669	668	0.15
	er-8n6	6.40E+07	8.00E+06	694	691	0.43
	er-9n6	7.20E+07	9.00E+06	NA	666	NA
Small-World	sw-n6	8.00E+06	1.00E+06	NA	1181708	NA
	sw-2n6	1.60E+07	2.00E+06	NA	2362921	NA
	sw-3n6	2.40E+07	3.00E+06	NA	3544153	NA
	sw-4n6	3.20E+07	4.00E+06	NA	4727802	NA
	sw-5n6	4.00E+07	5.00E+06	NA	5907930	NA
	sw-6n6	4.80E+07	6.00E+06	NA	7086309	NA
	sw-7n6	5.60E+07	7.00E+06	NA	8270077	NA
	sw-8n6	6.40E+07	8.00E+06	NA	9450576	NA
	sw-9n6	7.20E+07	9.00E+06	NA	10634374	NA

Table 4.2 – *Continued on next page*

Table 4.2 – *Continued from previous page*

Graph				$\lvert ECC\rvert - \lvert trivial\_cliques\rvert$		Relative
				Conte-Method	EO-ECC	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100\ \%$
Social Network	com-Amazon	9.26E+05	3.35E+05	186345	186330	0.01
	com-DBLP	1.05E+06	3.17E+05	153522	152094	0.93
	soc-youtube-snap	2.99E+06	1.13E+06	415060	415060	0.00
	com-LiveJournal	3.47E+07	4.00E+06	N/A	6538414	N/A

Table 4.2 shows the comparative results between `Conte-Method` and `EO-ECC`. We see there is no negative value in "Relative Difference" column. That means `Conte-Method` produces more number cliques that have a size of more than two. Table 4.1 showed `Conte-Method` gives a larger clique cover than `EO-ECC`, and now we have shown that their clique cover has more nonzeros than `EO-ECC`. Table 4.2 shows *N/A* in some fields. It means `Conte-Method` could not find an edge clique cover for those instances. We ran `Conte-Method` in Compute-Canada machine (graham) with 500 Gigabyte memory, and for seven days, and then we reported *N/A* if we did not get result or got an error.

4.4.2 Covering Index

Minimizing the space occupied by the solution is another measure to find redundancy. We measure the covering index concerning how many times a vertex or edge is present in cliques of the cover. As our `EO-ECC` algorithm, in most of the cases, find larger-sized cliques than `Conte-Method`, we see a higher covering index for using `EO-ECC`. Here we report the result of `EO-ECC` without using the post-processing method for assignment minimum edge clique cover (Algorithm 4).

Let, C be a clique cover of size k for graph $G = (V, E)$, where $|V| = n$ is the number of vertices and $|E| = m$ is the number of edges.

Mean Vertex Covering

The size of cliques in terms of vertices can be defined as, $\sum_{l=1}^{k} |C_l|$ and therefore, the average vertex covering index is $\frac{\sum_{l=1}^{k} |C_l|}{n}$. In the 27.6% of the cases, `EO-ECC` and `Conte-Method` have equal average vertex covering. `Conte-Method` for the rest of the cases, achieves less mean vertex covering than `EO-ECC`, and on average, this value is 3.4% smaller than that of `EO-ECC`.

Mean Edge Covering

The mean edge covering index is equal to $\frac{\sum_{l=1}^{k} |C_l|.|C_l-1|}{2.m}$. In 54 out of 197 cases, `EO-ECC` and `Conte-Method` have equal average edge covering. For the rest of the cases (72.4%), `Conte-Method` achieves less mean edge covering than `EO-ECC`, and on average, this value is 12% smaller than that of `EO-ECC`.

4.4.3 Runtime Analysis of EO-ECC

The performance comparison between `Conte-Method` and `EO-ECC` is shown in Figure 4.8. For this comparison, we consider the time required to find edge clique cover and the post-processing method to remove redundant cliques. Because Conte's algorithm dealt with removing redundant clique and therefore, for a fair comparison, we included that time too. Conte did not consider assignment minimum clique cover, so we separately show the runtime analysis for that post-processing method.

For fifteen DIMACS10 instances and 182 Erdős-Rényi, Small-World instances, we draw a cross, which is the ratio between the time needed by `Conte-Method` and `EO-ECC`, as a function of the number of the edges. The horizontal green line at height 10^0 means that `Conte-Method` took the same time as `EO-ECC` to process the corresponding graph, and a point at height 10^1 means that `Conte-Method` was ten times slower.

The graph processing rate is one of the quality assessment metrics for an algorithm. We report the processing rate of our algorithm for a selection of real-world (DIMACS10, SNAP) and synthetically generated (Erdös-Rényi, Small World) graphs in Table 4.3. Ta-

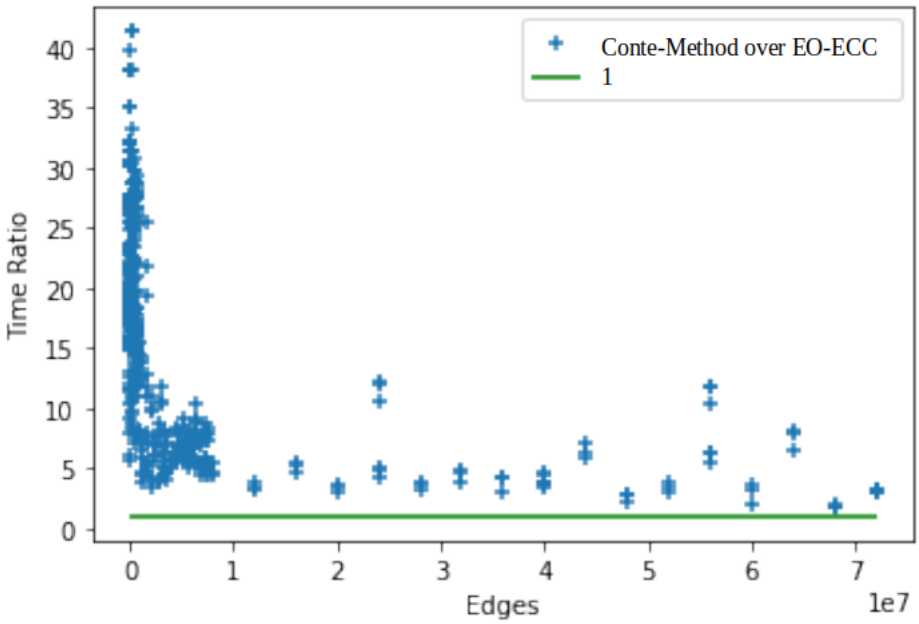


Figure 4.8: Ratio between the time used by `Conte-Method` and `EO-ECC` for each graph, as a function of the number of the edges (y-axis is in log-scale)

Table 4.3: Graph Processing Rate (Number of Edges Processed per Sec)

Group	Total instances	Largest rate	Smallest rate	Average rate
DIMACS10	15	$2.7E6$	$3.0E5$	$1.7E6$
SNAP	9	$2.5E6$	$6.2E4$	$1.5E6$
Erdős-Rényi	110	$2.0E6$	$1.2E5$	$8.9E5$
Small World	72	$1.7E6$	$4.3E5$	$1.1E6$

ble 4.3 shows the largest rate, the smallest rate, and the average rate for each set of graph instances. On DIMACS10 instances, the algorithm performs the best, while on Erdös-Rényi instances, the algorithm is not as efficient. This can be explained by the structural properties of graphs. Real-life and Small World synthetic instances display a power-law degree distribution resulting in a large proportion of vertices with very small degrees. Thus, the set intersection operation in our algorithm can be very efficient on those types of graphs.

The time needed by `Conte-Method`, `EO-ECC-D`, `EO-ECC-L`, and `EO-ECC-I` to process the corresponding graph from the DIMACS10 group is reported in Table 4.4. We observe that for all cases, `Conte-Method` took more time than `EO-ECC`.

Table 4.4: Test Results (run-time) for DIMACS10 matrices

Graph			Time in seconds			
Name	m	n	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
chesapeake	170	39	0.027	0	0	0
delaunay_n10	3056	1024	0.126	0	0	0.01
delaunay_n11	6127	2048	0.157	0.01	0	0
delaunay_n12	12264	4096	0.27	0.01	0	0.01
delaunay_n13	24547	8192	0.381	0.01	0.01	0.01
delaunay_n14	49122	16384	0.544	0.03	0.03	0.02
delaunay_n15	98274	32768	1.09	0.06	0.05	0.05
delaunay_n16	196575	65536	2.305	0.12	0.11	0.11
delaunay_n17	393176	131072	3.772	0.25	0.23	0.21
delaunay_n18	786396	262144	7.067	0.49	0.46	0.43
com-DBLP	1049866	317080	7.382	1.26	1.12	1.02
belgium_osm	1549970	1441295	14.569	0.75	0.67	0.57
delaunay_n19	1572823	524288	11.144	1.01	0.95	0.87
delaunay_n20	3145686	1048576	21.046	2.04	1.98	1.78
delaunay_n21	6291408	2097152	35.904	4.04	3.9	3.47

Complexity

Theorem 4.13 shows that, for an undirected and connected graph $G = (V,E)$, where $|V| = n$, and $|E| = m$, Algorithm 2 takes $O(m + \sum_{i=1}^{k} \frac{\rho_i \times (\rho_i - 1)}{2})$ time to find edge clique cover $\mathbb{C} = \{C_1, \ldots, C_k\}$, where ρ_i denotes the size of clique C_i and $(k \leq m)$.

Here the runtime depends on the number of edges and the size of each clique of the clique cover. However, we want to see the relationship between the total time (used to compute clique cover by `EO-ECC`) and the number of edges (m) for all the groups of our test instances.

Figure 4.9 shows the time used to compute clique covers by `EO-ECC`, where the time is a function of the number of edges in the graph. We consider the time in microseconds. A dot (x,y) states that the graph has x edges, and the algorithm spent y microseconds to finish the computation. We observe that the dots align with a line ($y = c \times x$), suggesting a linear running time of the algorithm, where c is a constant.

The runtime of the `EO-ECC` algorithm depends on the size of the cliques due to the *Find-CommonNeighbors* function that we used to grow the clique for an uncovered edge. *Find-CommonNeighbors* function does a merge operation between two sorted lists. So, the cost of using this function (merge cost), for two given sorted lists L_1, and L_2 is $max\{|L_1|, |L_2|\}$. Therefore, we also want to see the edge vs. merge cost relationship for using the *FindCom-*

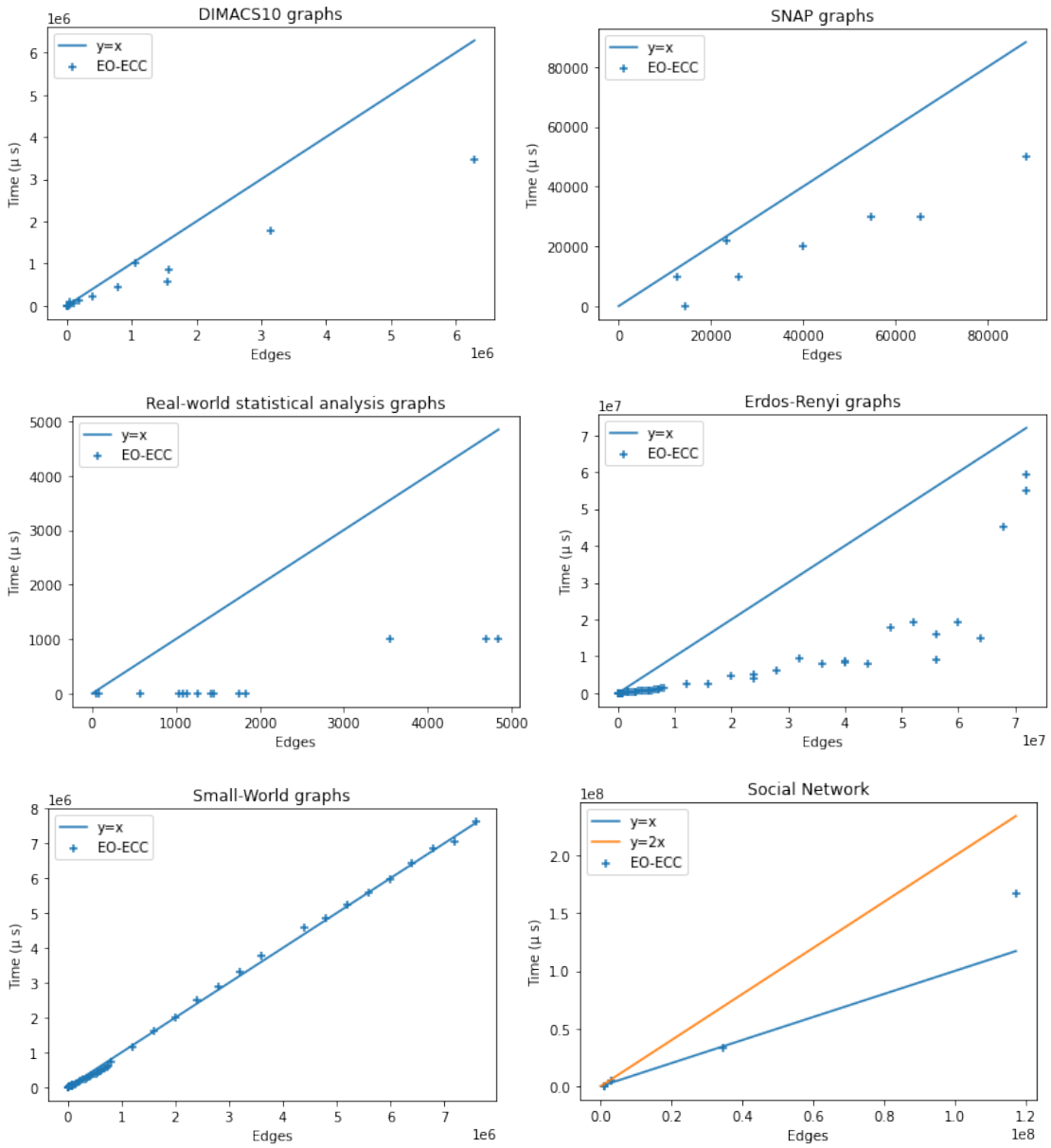


Figure 4.9: Runtime to find clique cover using `EO-ECC` and to remove redundant cliques using post-processing method

monNeighbors function.

In Figure 4.10, a dot (x, y) states that the graph has x edges, and the total merge cost for using *FindCommonNeighbors* is y. We observe that the dots align with a line ($y = c \times x$), suggesting the cost of finding common neighbors is linear, where c is a constant.

4.4.4 Analysis of the Assignment Minimum Edge Clique Cover

After we find a clique cover using `EO-ECC`, we use our proposed post-processing method to minimize the number of nonzeros to store the cover, creating an assignment minimum edge clique cover. The goal of the assignment-minimum-cover is to minimize the number of individual assignments of vertices (number of nonzeros) to cliques. Ennis et al. [23] presented a similar idea to get assignment minimum edge clique cover. However, their backtracking algorithm becomes costly for large graphs, especially when they have many maximal cliques.

Relative Difference in Number of nonzeros

We compare the total number of nonzeros before and after the post-processing method (Table 4.5). Ennis et al. [23] presented their result for some real-world graphs (triticale and wheat). We got the same assignment minimum cover for those instances. Table 4.5 shows that our post-processing method reduces the total number of nonzeros significantly.

Table 4.5: Relative Difference in Number of Nonzeros to Store Clique Cover by using and without using Post-processing (Assignment Minimum)

Graph				Number of nonzeros		Relative
				Unprocessed	Processed	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100$ %
	chesapeake	170	39	241	209	13.28
	delaunay_n10	3056	1024	3839	3548	7.58

Table 4.5 – *Continued on next page*

Table 4.5 – *Continued from previous page*

Graph				Number of nonzeros		Relative
				Unprocessed	Processed	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100\ \%$
DIMACS10	delaunay_n11	6127	2048	7656	7100	7.26
	delaunay_n12	12264	4096	15331	14218	7.26
	delaunay_n13	24547	8192	30719	28473	7.31
	as-22july06	48436	22963	85732	80055	6.62
	delaunay_n14	49122	16384	61486	56971	7.34
	delaunay_n15	98274	32768	122993	113920	7.38
	delaunay_n16	196575	65536	245386	227497	7.29
	delaunay_n17	393176	131072	491477	455409	7.34
	delaunay_n18	786396	262144	983203	910904	7.35
	com-DBLP	1049866	317080	806486	775868	3.80
	belgium_osm	1549970	1441295	3092786	3092780	0.00
	delaunay_n19	1572823	524288	1966239	1821680	7.35
	delaunay_n20	3145686	1048576	3933219	3643890	7.36
	delaunay_n21	6291408	2097152	7863444	7286128	7.34
SNAP	as-735	12572	7716	21724	20741	4.52
	ca-GrQc	14484	5242	11295	10931	3.22
	ca-HepTh	25973	9877	27341	26117	4.48
	Oregon-1	23409	11492	40374	37655	6.73
	p2p-Gnutella04	39994	10879	77904	77830	0.09
	p2p-Gnutella24	65369	26518	128413	128358	0.04
	p2p-Gnutella25	54705	22687	107523	107482	0.04

Table 4.5 – *Continued on next page*

Table 4.5 – *Continued from previous page*

Graph				Number of nonzeros		Relative
				Unprocessed	Processed	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100\ \%$
	p2p-Gnutella30	88328	36682	173204	173056	0.09
	wiki-Vote	100762	8297	208486	154581	25.86
Real-World	triticale1	55	13	23	20	13.04
	triticale2	86	17	42	32	23.81
	rapeseed1	576	47	314	215	31.53
	rapeseed2	1040	57	451	300	33.48
	rapeseed3	1260	64	567	344	39.33
	rapeseed4	1085	62	384	268	30.21
	rapeseed5	1456	64	561	358	36.19
	rapeseed6	1416	70	684	413	39.62
	rapeseed7	1758	74	756	476	37.04
	rapeseed8	1128	59	369	231	37.40
	rapeseed9	1835	76	902	565	37.36
	wheat1	4847	124	2320	1337	42.37
	wheat2	4706	121	2209	1316	40.43
	wheat3	3559	97	1480	856	42.16
Small-World	sw-n6	7999975	1000000	10790377	9844929	8.76
	sw-2n6	15999981	2000000	21573978	19683409	8.76
	sw-3n6	23999985	3000000	32359873	29524152	8.76
	sw-4n6	31999984	4000000	43147212	39367809	8.76
	sw-5n6	39999980	5000000	53933287	49208233	8.76

Table 4.5 – *Continued on next page*

Table 4.5 – *Continued from previous page*

Graph				Number of nonzeros		Relative
				Unprocessed	Processed	Difference
Group	Name	m	n	a	b	$\frac{(a-b)}{a} \times 100$ %
	sw-6n6	47999982	6000000	64710048	59041147	8.76
	sw-7n6	55999984	7000000	75519359	68906926	8.76
	sw-8n6	63999976	8000000	86305387	78742474	8.76
	sw-9n6	71999983	9000000	97087183	88579564	8.76
Erdős-Rényi	er-n6	8000000	1000000	15997821	15997821	0
	er-2n6	16000000	2000000	31997939	31997937	6.25E-06
	er-3n6	24000000	3000000	47997975	47997975	0
	er-4n6	32000000	4000000	63998145	63998144	1.56E-06
	er-5n6	40000000	5000000	79998081	79998079	2.50E-06
	er-6n6	48000000	6000000	95997937	95997936	1.04E-06
	er-7n6	56000000	7000000	111998062	111998060	1.79E-06
	er-8n6	64000000	8000000	127998017	127998013	3.13E-06
	er-9n6	72000000	9000000	143998082	143998080	1.39E-06
Social	com-Amazon	925872	334863	1266929	1169319	7.70
	com-DBLP	1049866	317080	806486	775868	3.80
	soc-youtube-snap	2987624	1134890	5477826	5041656	7.96
	com-LiveJournal	34681189	3997962	54614110	44062611	19.32

Runtime Analysis of the Post-processing Method for Assignment Minimum ECC

We have presented an algorithm, `AM-ECC` (Algorithm 4), to get assignment minimum edge clique cover for the given cover. Lemma 4.15 shows that `AM-ECC` takes $O(\sum_{i=1}^{k'} \rho_i + \frac{\rho_i \times (\rho_i - 1)}{2})$ time, where k' is the number of cliques (after removing the redundant cliques

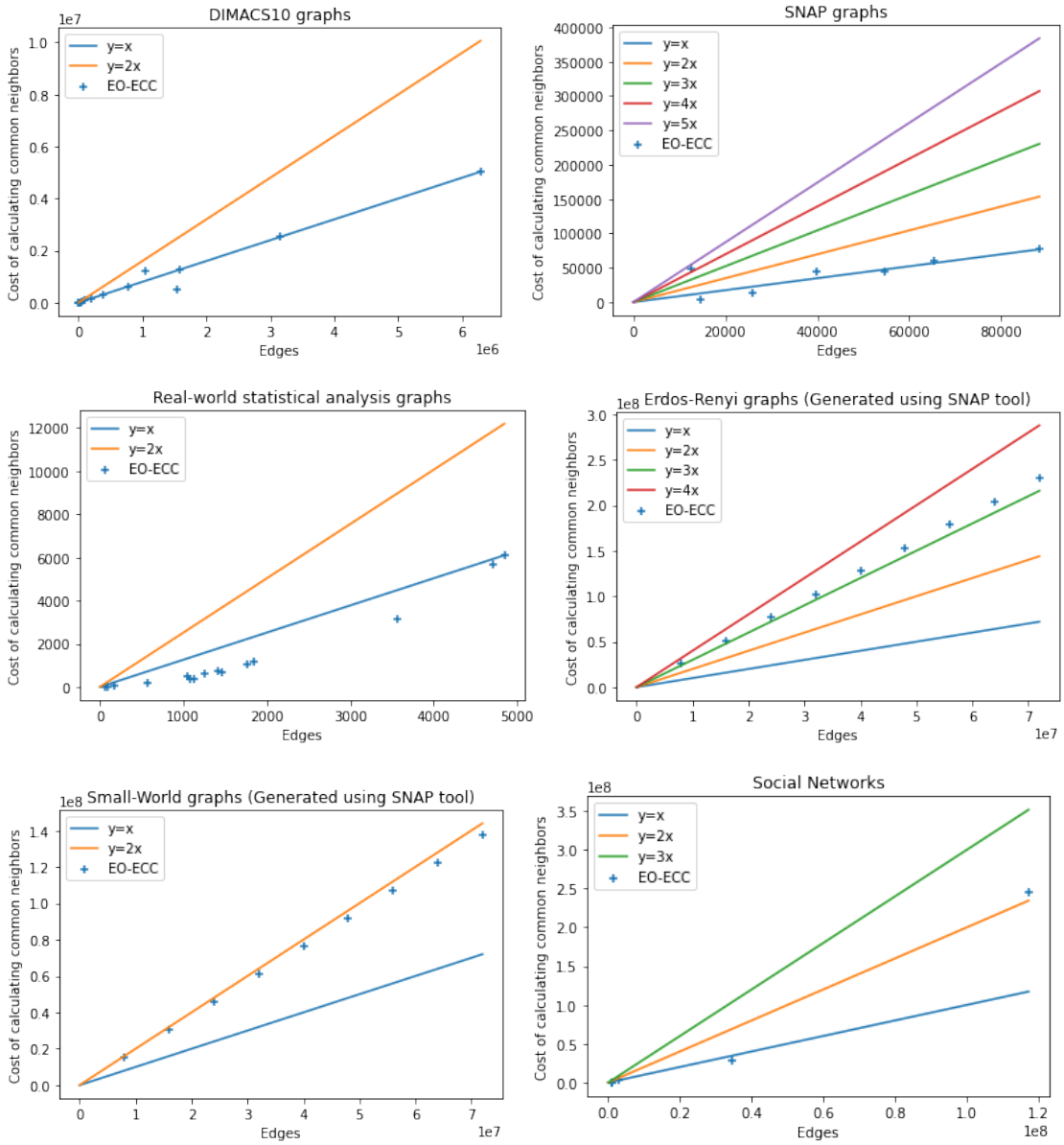


Figure 4.10: Total merge cost for using *FindCommonNeighbors* function

from the cover), and ρ_i is the size of clique C_i.

This lemma denotes that the runtime of the AM-ECC algorithm depends on the size of the edge clique cover because we test all the edges of each clique. We store our clique cover using an intersection matrix. Therefore, the cost for the AM-ECC algorithm is to find the edge index for the given two vertices. Assume we are testing clique C_i, where C_i is a clique of ECC, $\mathbb{C}$. Now, for two vertices u and v, where $u, v \in C_i$, we need to find out their edge index. We have a function *GetRowIndex* that returns the edge index of the input graph for given two vertices. Note, the input graph is stored using a column intersection matrix, where each column represents a vertex and each row represents an edge. This function intersects two columns corresponding to given vertices to get an edge index (row index). While we examine all the edges of a clique cover, for two given vertices, we might call the function *GetRowIndex* more than once. Therefore, we want to see how many intersection operations are done for the calling *GetRowIndex* function to test all the edges of the clique cover by plotting the edge vs. cost graph.

In Figure 4.11, a dot (x, y) states that the graph has x edges, and the total intersection operation cost for using *GetRowIndex* function is y. We observe that the dots align with a line ($y = c \times x$), suggesting the cost of AM-ECC is linear, where c is a constant.

4.5 Parallel Implementation of EO-ECC

In this section, we describe the parallel implementation of our algorithms. Alabandi et al. [3] proposed an algorithm that increases the parallelism without affecting the coloring quality and performs better than other parallel-coloring algorithms. Most of the parallel graph coloring algorithms [11, 15, 34, 77] followed the Jones-Plassmann approach [41], where Alabandi et al. included two shortcut techniques to improve the performance. Unfortunately, there is no parallel implementation for the edge clique cover algorithms. However, these coloring algorithms inspired us for the parallel implementation of our ECC algorithm. Now the question is whether it is possible to get perfect speed-up solving ECC. Indeed, we

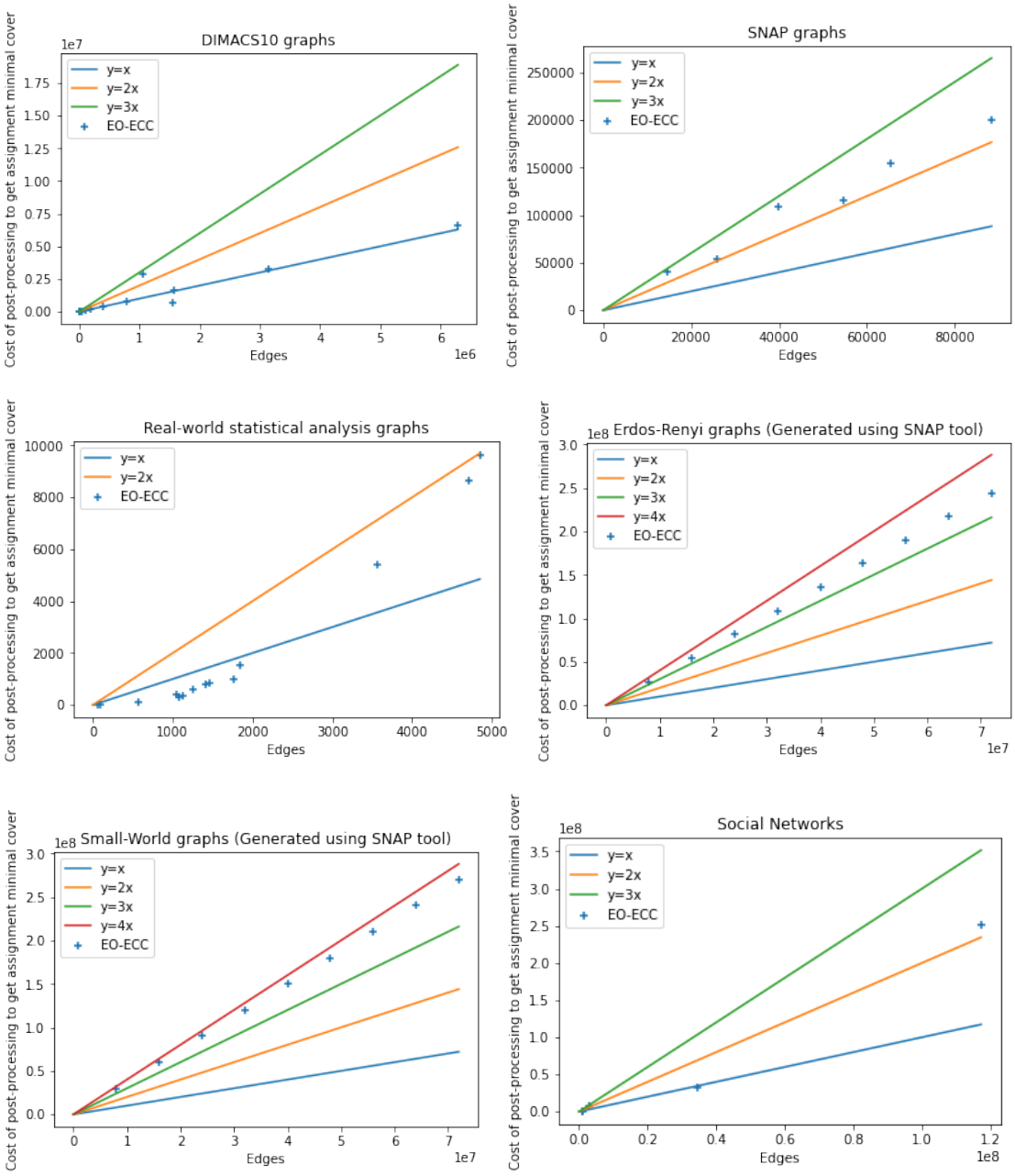


Figure 4.11: Total intersection cost for using *GetRowIndex* function in AM−ECC

can not tell, but we can try to parallelize the sub-methods used in `EO-ECC` algorithm, such as *FindNeighbors*, and *FindCommonNeighbors*.

This section targets parallel copy and merges algorithms because finding neighbors of a vertex copies a portion of our intersection matrix into an array. Finding the common neighbors set is related to merging two-sorted lists. We discussed the *intersection matrix* in previous chapters. Finally, we evaluate our `EO-ECC` algorithm with this parallel *FindNeighbor*, and *FindCommonNeighbor*.

OpenCilk [75] is considered as one of the powerful tools for optimizing parallelized algorithms on the CPU, an open-source platform to support Cilk multithreaded programming. It provides dynamic-analysis tools that can detect race conditions and can plot a speed-up graph for available threads. For these unique features, we have chosen OpenCilk over CUDA or OpenMP.

Section 4.5.1 describes the OpenCilk. After that, Section 4.5 recaps the serial implementation of the `EO-ECC` algorithm, and Section 4.5.3 describes the parallel implementations using OpenMP. Afterward, Section 4.5.4 explains the results.

4.5.1 OpenCilk

Cilk, developed by MIT in 1990, is a Parallel Asynchronous Many-Tasking System that aims towards extending the ordinary serial programming paradigm in a user-friendly manner. Compared to other concurrency platforms, Cilk is distinguished by its simplistic design and implementation while enabling big multithreading performance based on proven mathematical foundations. Then the same team from MIT developed OpenCilk in 2019 and has taken on the goal of supporting and delivering the language to the High-Performance Computing community via an open-source project. The openCilk initiative plays a crucial role in next-generation multicore research. OpenCilk [75] is a full-featured, open-source implementation of Cilk designed to run on Linux and other Unix-like systems.

Keywords

OpenCilk enables inherent parallelism to both C and C++. It introduces three simple but powerful keywords within its environment.

- **cilk_spawn:** Denotes that the right-hand side of the expression runs in parallel with the following statements in a fork-join execution mode.
- **cilk_sync:** Blocks the execution until all children of the current task block finish.
- **cilk_for:** Analogous to the standard C/C++ for loop keyword, permits loop iterations to run in parallel. Each iteration of a cilk_for loop is a separate strand that executes asynchronously.

By combining keywords *cilk_spawn*, and *cilk_sync*, the user can form complex sequences that compose non-trivial algorithms as we also have taken advantage of these to model parallel EO-ECC algorithm.

Reducer

OpenCilk provides most out-of-the-box reduction operations that aid in a large spectrum of algorithmic problems.

- **op_list_append:** Creates a list by appending elements at the back.
- **op_list_prepend:** Creates a list by appending elements at the front.
- **op_max:** Finds the maximum value over a set of values.
- **op_min:** Finds the minimum value over a set of values.
- **op_add:** Performs a reduced summation or subtraction over a set of elements.
- **op_string:** Creates string by appending characters.
- **op_vector:** Creates a vector by appending elements at the back.

- **op_and:** Performs the AND bitwise operations with reduction.
- **op_or:** Performs the OR bitwise operations with reduction.
- **op_xor:** Performs the XOR bitwise operations with reduction.

OpenCilk has a reducer class template by which user can create their reducers. In this work, we have used reducer *op_vector* while creating the common neighbor set by using a parallel set-intersection operation (similar to merge) on two sorted neighbor lists.

Applications

Example: Copy a list into another list

Let us have a list of elements. Now copying a given list into another is a widely used but rather demanding one in terms of memory utilization.

```
int main()
{
    vector<int>given_list = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};

    cilk::reducer<cilk::operator<int>> target_list;
    cilk_for(int i=0;i<given_list.size();i++)
    {
        target_list->push_back(given_list[i]);
    }
    return 0;
}
```

Figure 4.12: Copy between lists using parallel loop

Now we can design two different parallel algorithms: (1) a parallel loop using keyword *cilk_for* and *append* reducer (see Figure 4.12), or (2) a recursive function using keywords *cilk_spawn*, and *cilk_sync*, and using *vector* reducer (see Figure 4.13).

Example: Merge operation

We have implemented the parallel merge algorithm in OpenCilk discussed by Cormen et al. [18] in their book "Introduction to Algorithms," Chapter 27, Section 3. The implementation

```
void copy(int l, int u, vector<int>& given_list , vector<int>& target_list){
    if(l==u)   target_list[l]=given_list[l];
    else if(l<u)
    {
        int mid = (l+u)/2;
        cilk_spawn copy(l,  mid,  given_list , target_list);
        cilk_spawn copy(mid+1, u, given_list , target_list);
        cilk_sync;
    }
}
int main(){
    vector<int>given_list = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    vector<int>target_list(given_list.size());
    int l = 0;
    int u = given_list.size()-1;
    copy(l,u,given_list,target_list);
    return 0;
}
```

Figure 4.13: Copy between lists using parallel recursive function

is shown in Figure 4.14. Cormen et al. showed parallelism of merge algorithm has work: $T_1(n) = O(n)$, and Span: $T_\infty = O(lg^2 n)$, where n is the length of the list.

In our EO-ECC algorithm, we can compute the list of neighbors of a vertex. Then we need to compute the common neighbors between two vertices by intersecting the neighbor lists. We used the same parallel merge technique to find the common neighbor between two vertices.

4.5.2 Edge Clique Cover Algorithm

At the beginning of this chapter, we discuss our "edge-centric" edge clique cover algorithm, `EO-ECC`, for solving edge clique cover problems using ordered edges. In this section, we recap `EO-ECC`, but our main focus is on two used sub-methods: *FindNeighbors*, and *FindCommonNeighbors*.

EO-ECC Algorithm

`EO-ECC` algorithm frequently needs to find out the neighbor sets of the vertices and find the common neighbor set between two vertices. The complete algorithm is discussed in Section 4.3.2. Case-II runs until W is empty, where W is the set of common neighbors.

```
// *C stores the merge result of given lists *A and *B
// na = |A|, nb = |B|
void P_Merge(int *C, int *A, int *B, int na, int nb)
{
    if (na < nb) {
        cilk_spawn P_Merge(C, B, A, nb, na);
    }
    else if (na==1) {
        if (nb == 0) {
            C[0] = A[0];
        }
        else {
            C[0] = (A[0]<B[0]) ? A[0] : B[0]; /* minimum */
            C[1] = (A[0]<B[0]) ? B[0] : A[0]; /* maximum */
        }
    }
    else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        // BinarySearch takes O(lg n) serial time
        // If parallel, then its span is also O(lg n)
        cilk_spawn P_Merge(C, A, B, ma, mb);
        cilk_spawn P_Merge(C+ma+mb, A+ma, B+mb, na-ma, nb-mb);
        cilk_sync;
    }
}
```

Figure 4.14: OpenCilk implementation of merge operation

Each iteration then finds the neighbor set of a vertex and updates the common neighbor set W. We also showed that this Case-II dominates the `EO-ECC` algorithm's run time. Therefore, we were motivated to parallelize these two sub-methods: *FindNeighbors*, and *FindCommonNeighbors*. Before explaining the parallel implementation, we need to visualize how we have implemented these methods with an example.

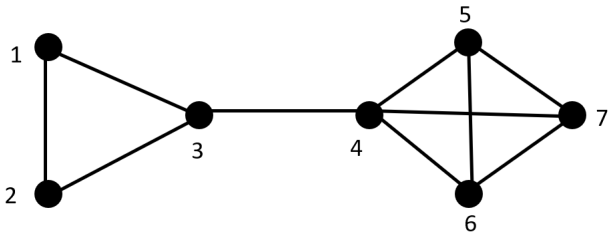
Serial Implementation

FindNeighbors

For an undirected graph G, we can define the neighbor set of a vertex $v \in V$ as follows.

$$Neighbor(v) = \{w \mid \{v,w\} \in E\}$$

Recall, we store the given graph using the column intersection matrix (see Section 3.2). In this column-intersection matrix X, the number of rows is $|E| = m$. Each row contains only two nonzeros: v, and another is the neighbor of v. Let, $d(v)$ is the degree of vertex v. Now, while finding the neighbor of vertex v, we only check the rows where v is present.



	1	2	3	4	5	6	7
1	x	x					
2	x		x				
3		x	x				
4			x	x			
5				x	x		
6				x		x	
7				x			x
8					x	x	
9					x		x
10						x	x

Figure 4.15: Given graph G and its intersection matrix

Example:

Given a graph $G = (V, E)$ consists of a set of vertices V, and a set of edges E, where $|E| = m$ and $|V| = n$. An intersection matrix associated with graph G is a matrix $X \in \{0,1\}^{m \times n}$ where for edge $e_l = \{v_i, v_j\}, l = 1, \ldots, m$ we have $X(l,i) = X(l,j) = 1$, and all other entries of matrix X are zero. Figure 4.15 represents a graph and its intersection matrix.

But with our sparse matrix representation we can store only the non-zero values of this intersection matrix using two $2m$ size array, one m size array, and one n size array (see Figure 4.16).

Let we need to find out the neighbor set of vertex 4. Figure 4.16 shows how we can find the neighbor set for a given vertex. First, from the Col_Ptr, we get the range of row indices of vertex 4, and the range i is from $Col\_Ptr[4]$ to $Col\_Ptr[5] - 1$. Second, we access $Row\_Ind[i]$ with these indices and get the rows where vertex 4 is present in the intersection matrix. Third, we have to find the columns of these rows to find the neighbors. Note, each row contains exactly two vertices (columns). One is vertex 4, and another one is 4's neighbor. Therefore, using Row_Ptr we get the range j of the indices of columns for row

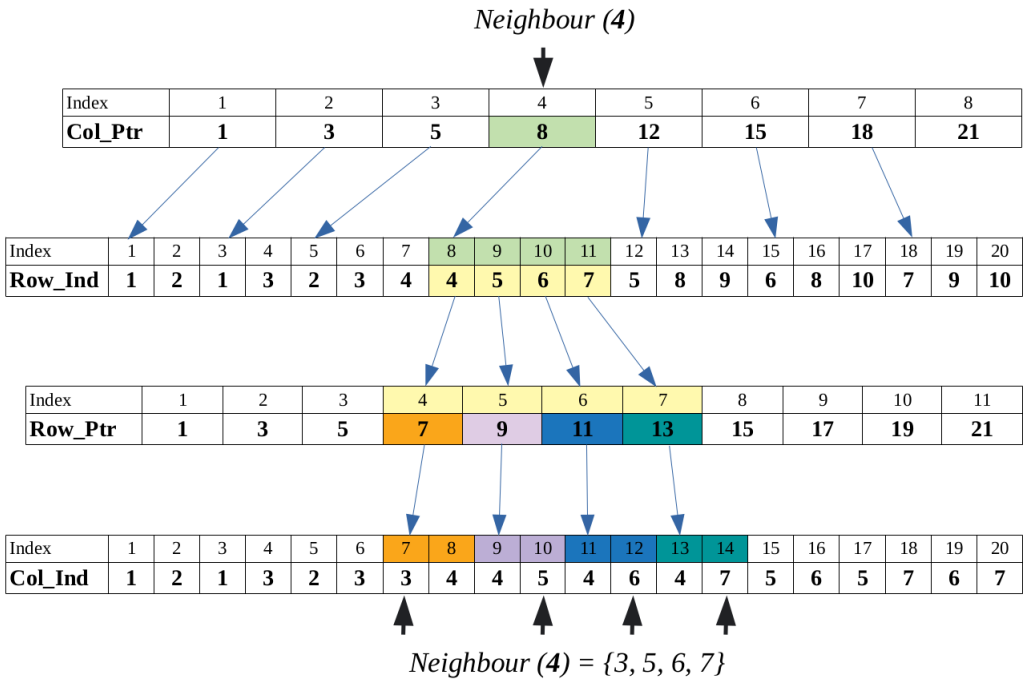


Figure 4.16: An example of finding a vertex's neighbor set from the intersection matrix

$Row\_Ind[i]$. Finally, $Col\_Ind[j]$, or $Col\_Ind[j+1]$ contains 4, and so we take the other vertex as a neighbor. We get, $Neighbor(4) = \{3,5,6,7\}$

In summary, we access a one-dimensional matrix of size $2m$, i.e., Col_Ind to find the neighbor set, where other arrays help map the intersection matrix. The job of finding a neighbor set was to copy some selected data from the Col_Ind matrix to the $Neighbor$ array.

FindCommonNeighbors

Let, we are given lists of neighbors for vertex v_1 and vertex v_2. To get neighbor list, we use *FindNeighbors(v)* method, where $v \in V$. We construct the set of neighbor in a way that it will be sorted (for example, see Figure 4.16). Now using *Intersect(L1,L2)* we can get the common neighbor of vertices v_1 and v_2.

We can find the set intersection, S, between these two lists, by merging $L1$ and $L2$. The time complexity is given by $O(max\{|L1|,|L2|\})$.

Example: Given a graph $G = (V,E)$ as shown in Figure 4.15. Let us have neighbor set

for vertices 4 and 5 using the *FindNeighbors* method. We now find the common neighbor set between vertices 4 and 5.

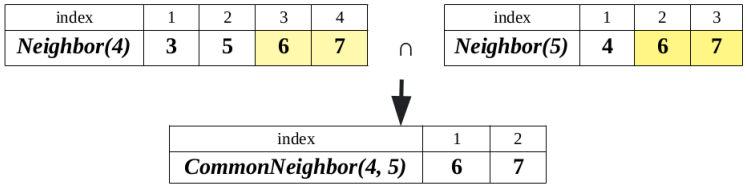


Figure 4.17: Common neighbor set between vertices 4 and 5

Like regular merge operation, we check all the elements of the smaller list and store only if it matches an element of the other list. Therefore, we get $CommonNeighbor(4,5) = \{6,7\}$.

4.5.3 Parallel Implementation on the OpenCilk

ParallelFindNeighbors

The parallel version of the *FindNeighbors* method works as a parallel algorithm to copy a given list as described in Section 4.5.1. This section illustrates the parallel version of the *FindNeighbors* method with an example.

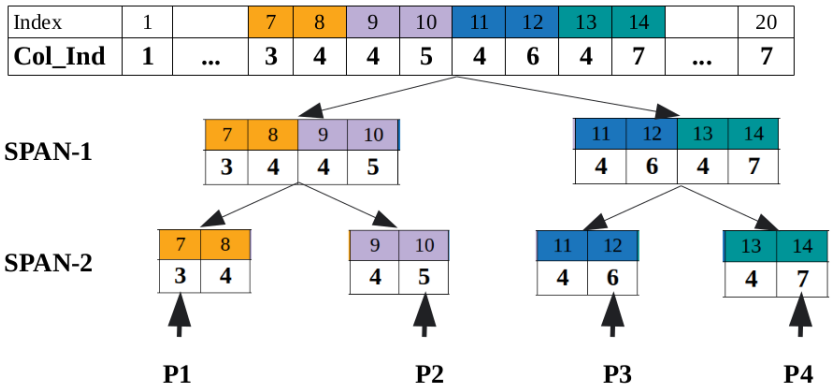


Figure 4.18: Find neighbor set of vertex 4 using parallel method

Figure 4.18 shows an example of finding the neighbor set of vertex 4 using the parallel method. In the first span, the list ($Col\_Ind[7] \ldots Col\_Ind[14]$) is divided into two sub-lists. Then span-2 divides both sub-lists into another two sub-lists, and therefore span-2 has four

sub-list, and those lists are undividable. This span copies four neighbors to the neighbor list simultaneously, assuming four threads are available. Practically, instead of dividing the lists until we get one neighbor, we set a *Threshold* value; below that value, the *FindNeighbors* method works sequentially. This technique helps to reduce overhead.

The parallel implementation of the *FindNeighbors* method in OpenCilk is shown in Figure 4.19.

```
void Matrix::neighbor_copy(int v1, int *Neighbors_v1, unsigned jl, unsigned ju, unsigned size)
{
    if(jl==ju)
    {
        int l = row_ind[jl];
        int k = 2*l-1;
        int j1 = col_ind[k];
        int j2 = col_ind[k+1];
        int pos=0;
        if(size==1) pos=jl;
        else        pos=(jl%size)+1;

        if(pos>=N) cout<<"POS: " <<pos <<endl;

        if(j1!=v1)  Neighbors_v1[pos]=j1;
        else        Neighbors_v1[pos]=j2;
    }
    else if((ju-jl)<Threshhold){  // if the number of neighbors is less thatn this "Threshhold" value,
                                  // then this calculation will be sequential
        unsigned mid = (jl+ju)/2;
        neighbor_copy(v1, Neighbors_v1, jl,    mid, size);
        neighbor_copy(v1, Neighbors_v1, mid+1, ju,  size);
    }
    else if(jl<ju)                // Otherwise, calculation is done by parallel recursive call
    {
        unsigned mid = (jl+ju)/2;
        cilk_spawn neighbor_copy(v1, Neighbors_v1, jl,    mid, size);
        cilk_spawn neighbor_copy(v1, Neighbors_v1, mid+1, ju,  size);
        cilk_sync;
    }
}

void Matrix::find_neighbors(int v1, int *Neighbors_v1)
{
    unsigned jl = jpntr[v1];    // jpntr is the Col_Ptr array
    unsigned ju = jpntr[v1+1]-1;
    Neighbors_v1[0]=ju-jl+1;    // number of neighbors of vertex v1
                                // Neighbors_v1 stores the neighbors of v1
    neighbor_copy(v1, Neighbors_v1, jl, ju,jl);
}
```

Figure 4.19: OpenCilk implementation of *FindNeighbors* method

ParallelFindCommonNeighbors

The parallel version of the *FindCommonNeighbors* method works as a parallel algorithm to merge two sorted lists as described in Section 4.5.1. This section illustrates the parallel version of the *FindCommonNeighbors* method with an example.

Figure 4.20 shows an example of finding the common-neighbor set between vertices 4 and 5 using the parallel method. The list, $Neighbor(4)$, is divided into two sublists in the first span. The mid-value of the list $Neighbor(4)$ is 5. Using binary search, we look for 5 in the list $Neighbor(5)$. Therefore the list $Neighbor(5)$ has two sub-lists: $\{4\}$ and $\{6,7\}$.

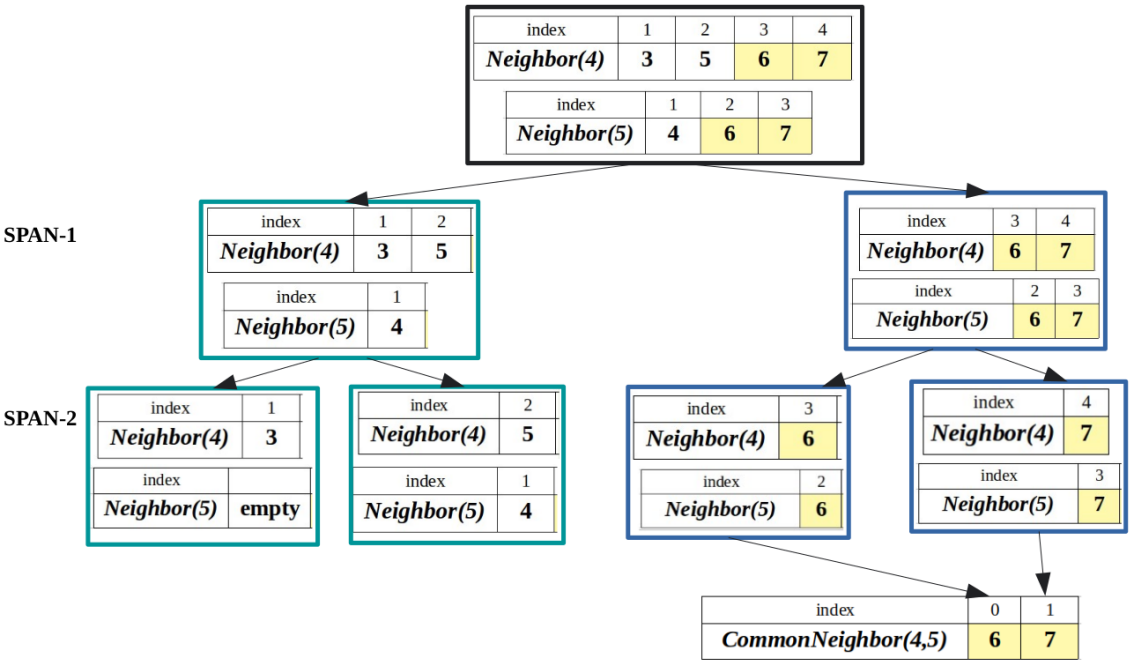


Figure 4.20: Find common neighbor between vertices 4 and 5 using parallel method

Using the same technique, span-2 divides sub-lists from span-1. After this span, we have four groups of lists to intersect. These groups can intersect two sublists simultaneously, assuming we have four threads. Finally, we get common neighbors 6 and 7. Practically, instead of dividing the lists until we get one element in the sublist, we set a *Threshold* value; below that value, the *FindCommonNeighbors* method works sequentially. This technique helps to reduce overhead.

The parallel implementation of the *FindCommonNeighbors* method in OpenCilk is shown in Figure 4.21.

4.5.4 Results

Test Environment

We use high-performance computing resources available at the Graham cluster (located at the University of Waterloo) of Compute Canada facility. A node at the Graham cluster consists of multiple 2.1 GHz Intel E5, E7, or Xeon CPUs, also referred to as threads, with memory varying from 124G to 3022G [12].

```
void p_intersect(int LL1,int UL1,int *L1,int LL2,int UL2,int *L2, cilk::reducer<cilk::op_vector<int>>& P)
{
    int len_L1= UL1-LL1+1;
    int len_L2= UL2-LL2+1;
    if(len_L2>len_L1)cilk_spawn p_intersect(LL2,UL2,L2, LL1,UL1,L1, P);
    else if(len_L1==1){
        if(L1[LL1]==L2[LL2]) P->push_back(L1[LL1]);
    }
    else if((len_L1-len_L2)<Threshold){    //Below this threshold the code is sequential
        int j=0;
        int mid_L1= (UL1+LL1)/2;
        int search_for = L1[mid_L1];
        int ret_status = binary_search(LL2, UL2, L2, search_for , &j);
        if(ret_status>0) j = ret_status;  // else we found a place at j
        if(j!=0){
             p_intersect(LL1, mid_L1 ,L1,  LL2, j, L2,   P);
             p_intersect(mid_L1+1,UL1,L1,  j+1,UL2,L2,   P);
        }
    }
    else{
        int j=0;
        int mid_L1= (UL1+LL1)/2;
        int search_for = L1[mid_L1];
        int ret_status = binary_search(LL2, UL2, L2, search_for , &j);
        if(ret_status>0) j = ret_status;    // else we found a place at j
        if(j!=0){
             cilk_spawn p_intersect(LL1, mid_L1 ,L1,  LL2, j, L2,   P);
             cilk_spawn p_intersect(mid_L1+1,UL1,L1,  j+1,UL2,L2,   P);
             cilk_sync;
        }
    }
}

void Matrix::find_common_neighbors(int *Neighbors_v1,int *temp_Neighbors)
{
    cilk::reducer<cilk::op_vector<int>> P;
    Common_Neighbors[0]=0;
    p_intersect( 1,Neighbors_v1[0],Neighbors_v1, 1,temp_Neighbors[0],temp_Neighbors, P);
    vector<int> Q;
    P.move_out(Q);
    Common_Neighbors[0]=Q.size();
    for (unsigned i = 0; i < Q.size(); i++) {
        Common_Neighbors[i+1]=Q[i];
    }
}
```

Figure 4.21: OpenCilk implementation of *FindCommonNeighbors* method

We compile our OpenCilk algorithms with optimization level 3 (O3 flag). We compile and run the code using OpenCilk's built-in LLVM, Clang, and Clang++. For each graph, we ran our code using threads (CILK_NWORKERS) 1 to 8.

Benchmarks

Extracting information from large graphs is challenging due to their sparse nature. We presented an efficient graph representation technique and proposed algorithms to solve the edge clique cover problem and triangle counting, which helps analyze graphs. However, the performance of our *FindNeighbors* and *FindCommonNeighbors* methods depends on the vertices' degree. From our previous analysis, we observed that even though some vertices

have a considerable degree, most of the vertices have a minimal degree. Therefore, to benefit from parallelism, we need a large degree for a reasonable amount of vertices. For this reason, we set a threshold value for both of our methods, below which the methods will work sequentially.

We test our parallel implementation for large sparse graphs, such as "delaunay" from the DIMACS10 group. The degree of the vertices of these "delaunay" graphs vary from 2 to 4; Therefore, we got no speedup.

To visualize the benefit of parallelism, we need graphs where the vertices have a large degree. Therefore, we generate graphs, $G = (V,E)$, where $|V| = n$, and $|E| = m$, such that each vertex has degree $d_v = \frac{n}{2}$. Therefore, $m = \frac{n^2}{4}$. With these attributes, we generated Random k-regular graphs using the Stanford Network Analysis Project (SNAP) [50].

Discussion

Table 4.6: Test Results for Parallel Processing

Graph				Time (sec) using thread							
Name	m	n	deg	1	2	3	4	5	6	7	8
GEN-1	32000000	8000	4000	437.802	221.39	150.5	115.04	99.6	78.25	70.1	59.52
GEN-2	4000000	4000	2000	109.51	55.85	38.29	29.267	24.918	21.6	19.35	15.806
GEN-3	1000000	2000	1000	27.748	13.979	10.097	7.724	6.943	6.01	5.69	5.1
GEN-4	250000	1000	500	6.887	3.587	2.624	2.073	1.955	1.901	1.789	1.67

Table 4.6 reports run time required by the generated test instances to run the `EO-ECC` algorithm with parallel *FindNeighbors* and *FindCommonNeighbors* methods varying number of threads from 1 to 8. Figure 4.22 shows the speedup for these instances. We observe that when the degrees of the vertices are higher (Gen-1, Gen-2, and Gen-3), we get a nearly perfect linear speedup. On the other hand, speedup is not achieved when we use more threads for instances having fewer vertex degrees(Gen-4).

OpenCilk heavily relies on the compiler to achieve optimizations, while its simplistic architecture encourages the user to achieve great results with minimal effort. The predefined set of reducers powerfully tackles the barriers introduced by locks and mutual exclusions

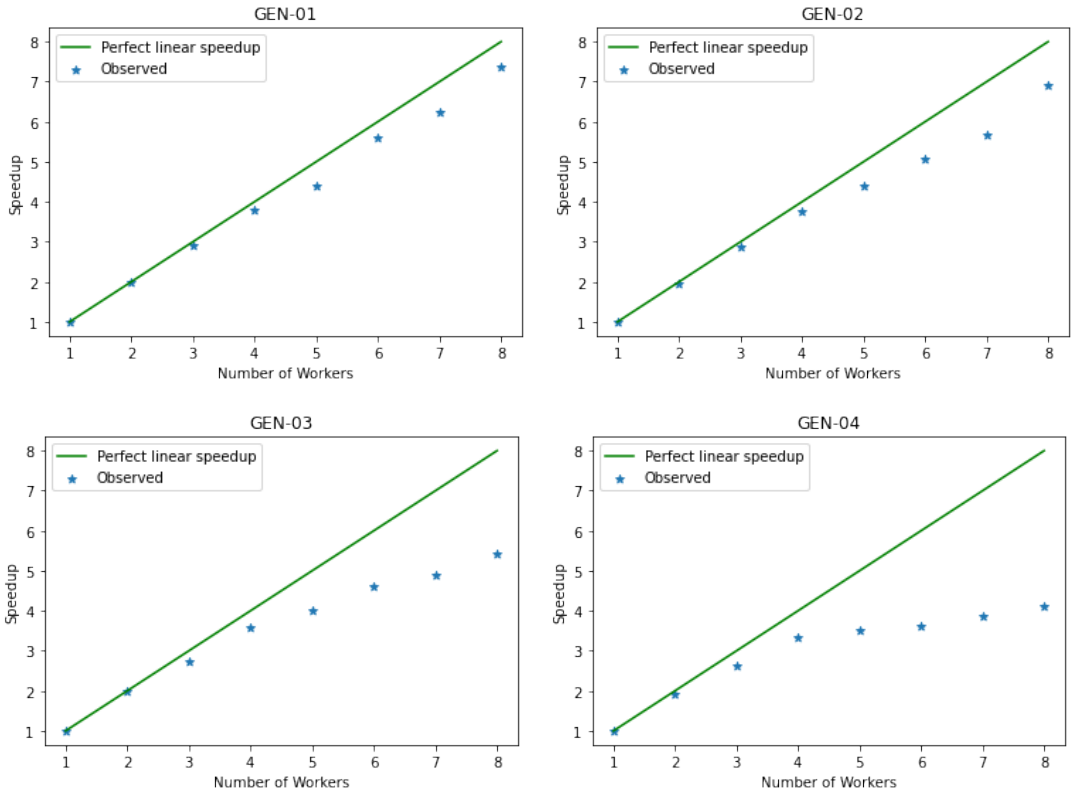


Figure 4.22: `EO-ECC` speedup

and provides extraordinary results in terms of performance. We can focus on parallelizing the entire edge clique cover algorithm for future research, including ordering the vertices and edges. OpenCilk has several reducers and holders other than three keywords: *cilk_spawn*, *cilk_sync*, and *cilk_for*. These additional features can help us with this further research.

4.6 Conclusion

In this chapter, we have proposed a compact representation of network data. The edge clique cover problem is recast as a sparse matrix determination problem. The notion of *intersection matrix* provides a unified framework that facilitates the compact representation of graph data and efficient implementation of graph algorithms. The adjacency matrix representation of a graph can potentially have many nonzero entries since it is the product of an intersection matrix with its transpose.

We have compared our results concerning the clique cover size and runtime with the current state-of-the-art algorithm for minECC [17]. For 219 test instances (from DIMACS10, SNAP, Real-World, Small-World, and Erdős-Rényi groups), where the number of edges varies between 170 and 7.6×10^7, our `EO-ECC` algorithm produces smaller or equal size clique covers than the `Conte-Method` [17]. `EO-ECC` is also significantly faster than the `Conte-Method`. `EO-ECC` algorithm runs in linear time, which allowed us to process extremely large graphs, both real-life and generated instances. Finally, our algorithm is highly scalable on large problem instances, while the algorithm of `Conte-Method` does not terminate on instances containing 7×10^7 or more edges within a reasonable amount of time.

A less well-studied but related problem, known as the *Assignment Minimum Edge Clique Cover* arising in computational statistics, is to minimize the number of individual assignments of vertices to cliques. It is not always possible to find assignment-minimum clique coverings by searching through those that are edge-clique-minimum. Ennis et al. [22] presented a post-processing method with an existing ECC algorithm to solve this problem.

However, their backtracking algorithm becomes costly for large graphs, especially when they have many maximal cliques. In this chapter, we have proposed an algorithm `AM-ECC`, where our edge-centric method with a post-processing step gives the assignment minimum cover calculation.

Chapter 5

Analyzing Large Complex Networks by Counting and Enumeration of Triangles

5.1 Introduction

The presence of triangles in network data has led to the creation of many metrics to aid in the analysis of graph characteristics. As such, the ability to count and enumerate these triangles is crucial to applying these metrics and gaining further insights into the underlying composition and distribution of these graphs. Generalizations aside, the applications of triangle counting are as ubiquitous as the triangles themselves, including transitivity ratio - the ratio between the number of triangles and the number of paths of length two in a graph - and clustering coefficient - the fraction of neighbors for a vertex i of a graph who are each other's neighbors. Other real-life applications of triangle counting include spam detection [4], network motifs in biological pathways [56], and community discovery [62]. However, before any network analysis can be undertaken, the underlying data structure of a graph must be critically examined and understood. An efficient representation of network data will dictate analysis capabilities and improve algorithm performance and data visualization potential [8]. Note that large real-life networks are typically sparse in nature, so efficient computations of these graphs must be able to account for their sparsity and skewed degree distribution [2]. A consistent structure makes linear algebra-based triangle counting methods appealing, and most methods use direct or modified matrix-matrix multiplication, with a notable exception being the implementation of Low et al. [53]. In this chapter, we propose an "intersection" representation of network data obtained as a list of edges [79] and

based on sparse matrix data structures [33]. Our triangle enumeration algorithm derives its simplicity and efficiency by employing matrix-vector product calculations as its main computational kernel. The local triangle count and edge support information are then acquired from the enumerated triangles obtained as the result of this matrix-vector multiplication.

5.1.1 k-count Distribution

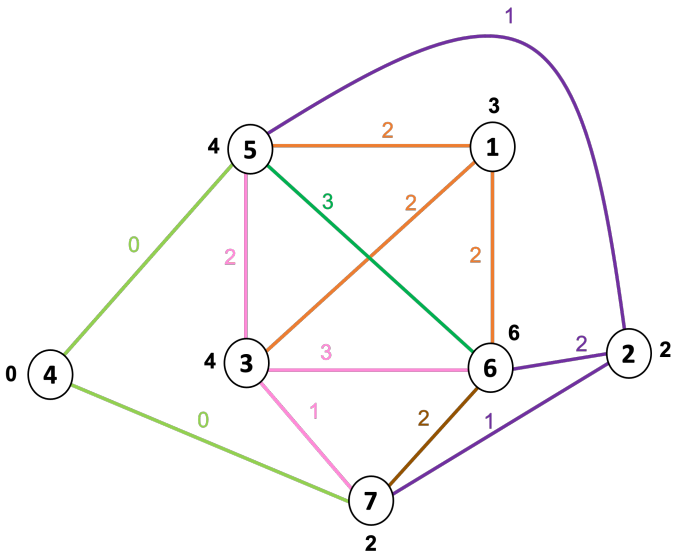
Application proxies provide a simple yet realistic way to assess the performance of real-life applications' architecture and design. Below, we outline the main components of the miniTri data analytics proxy [83], which we use to demonstrate the effectiveness of our intersection-based graph representation and computation.

Consider an undirected graph, $G = (V,E)$. For $v \in V$ a path of length 2 through v is a sequence of vertices $u - v - w$ such that $e_1 = \{u,v\} \in E$ and $e_2 = \{v,w\} \in E$. Such a length-2 path is termed a *wedge* at vertex v. Let $d(v)$ denote the number of edges *incident* on v, also defined as the number of vertices x such that $\{v,x\} \in E$, or the degree of v. The number of wedges in G is then given by $\sum_{v \in V} \binom{d(v)}{2}$. A wedge $u - v - w$ is a *closed wedge* or a *triangle* if $e_3 = \{v,w\} \in E$. Let $\delta(v)$ and $\delta(e)$ denote the number of triangles incident on vertex v and edge $e = \{u,v\}$ respectively. In the literature $\delta(v)$ is known as the *local triangle count* or *triangle degree* of vertex v and $\delta(e)$ is known as the *support* or *triangle degree* of edge $e = \{u,v\}$. We denote by $\Delta(G)$ the number of triangles contained in graph G. Since a triangle is counted at each of its three vertices, we have $\Delta(G) = \frac{1}{3}\sum_{v \in V} \delta(v)$. Let $H = (V',E')$ be a subgraph of G where $|V'| = k$ and each pair of vertices are connected by an edge (H is a k-clique). Then H contains $\binom{k}{3}$ triangles and $\delta(v) = \binom{(k-1)}{2}$ and $\delta(e) = (k-2)$ for $v \in V'$ and $e \in E'$. Let t be a triangle in G and let $\delta(t_x) = \min_x \delta(x)$, where x is a vertex of t and $\delta(t_e) = \min_e \delta(e)$ where e is an edge of t. The *k-count* of triangle t is defined to be the largest k such that

1. $\delta(t_x) \geq \binom{(k-1)}{2}$ and

2. $\delta(t_e) \geq (k-2)$

The main computational task of miniTri is to compute the k-count distribution of the triangles of an input graph. Figure 5.1 displays an example input graph with 7 vertices and 13 edges. Each vertex i is circled and contains a label that represents its identity. Beside each vertex i is an integer denoting its local triangle count $\delta(i)$, and there is an integer beside each edge $e = \{i, j\}$ denoting its support $\delta(e)$. The graph contains 7 triangles. The table of Figure 5.1 enumerates the triangles in the graph and displays the local triangle count and support of the vertices and edges together with the k-count of the triangles. Each row of the table lists the vertex labels of a triangle followed by the local triangle count, support, and k-count. There are 4 triangles with k-count value 4 and 3 triangles with k-count value 3. Let ω be the size of the largest clique in G. Then the graph contains at least $\binom{\omega}{3}$ triangles with k-count value of at least ω. Therefore, the k-count distribution can be used to obtain a lower bound on the size of the largest clique of a graph.



5.1.2 Triangle Centrality and Ranking

Triangle centrality helps find important vertices in a graph basing the concentration of triangles surrounding each vertex. An important vertex in triangle centrality is at the center of many triangles [9]. Such vertex may be present in many triangles.

u	v	w	$\delta(u)$	$\delta(v)$	$\delta(w)$	$\delta(e_1)$	$\delta(e_2)$	$\delta(e_3)$	K-Count
1	3	5	3	4	4	2	2	2	4
1	3	6	3	4	6	2	2	3	4
1	5	6	3	4	6	2	2	3	4
2	5	6	2	4	6	1	2	3	3
2	5	7	2	6	2	2	1	2	3
3	5	6	4	4	6	2	3	3	4
3	6	7	4	6	2	3	1	2	3

Figure 5.1: k-count table for the example input graph

Consider an undirected graph G. Let $N(v)$ be the neighborhood set of v, $N_\Delta(v)$ is the set of neighbors that are in triangles with v, and $N_\Delta^+(v)$ is the closed set that includes v. $\Delta(v)$ denotes the local triangle count of vertex v, and $\Delta(G)$ denotes the total triangle count of the graph G. Recently, Burkhardt [9] introduced triangle centrality as a new centrality measure that captures the influence of triangles on the importance of vertices. The triangle centrality for v is given by

$$TC(v) = \frac{\frac{1}{3}\sum_{u \in N_\Delta^+(v)} \Delta(u) + \sum_{w \in \{N(v) \setminus N_\Delta(v)\}} \Delta(w)}{\Delta(G)}$$

By definition, the centrality values indicate the proportion of triangles centered at a vertex bounded in the range $[0,1]$.

Li and Bader [51] presented a rapid implementation of triangle centrality using GraphBLAS [45], an API specification for describing graph algorithms in the language of linear algebra. This chapter implements triangle centrality and presents a comparative result followed by a parallel salable version.

5.1.3 Organization of the Chapter

The remainder of the chapter is organized as follows. In Section 5.2, we introduce the notion of the intersection representation of network data and our data structure, followed by a brief description of the triangle enumeration, the *k*-count algorithm, and triangle centrality calculation. The rest of the section explains the main ideas in our intersection matrix-based triangle enumeration using an illustrative example. Section 5.3 outlines the computing environment employed to perform numerical experiments and presents triangle enumeration results on three sets of representative network data. miniTri [83] and its successor, which we call *miniTri2*, are the reference implementations by which we present comparative running times and demonstrate that our method scales very well on massive network data, and can be very flexible in its extensions to the analysis of network characteristics such as Truss Decomposition [14] and triangle ranking [9]. In Section 5.4, we describe the parallel implementation of our intersection algorithm, **fullCount** for triangle count, enumeration, *k*-count, and triangle centrality. First, we describe the parallel triangle counting algorithm. Then we discuss the parallel algorithm for our **fullCount** algorithm concerning triangle count, local triangle count, and *k*-count. Finally, we present a parallel version of our triangle centrality calculation method. A shared memory parallel implementation of our algorithm using OpenMP is being developed. We observe reasonable speedups using multiple threads. The chapter is summarized in Section 5.5 with pointers on future research directions.

5.2 Intersection Representation of Network Data

We used *intersection matrix* (discussed in Section 3.2), to store the given undirected graph. This matrix requires a row for each edge. Figure 5.2 shows the intersection matrix representation of the example input graph.

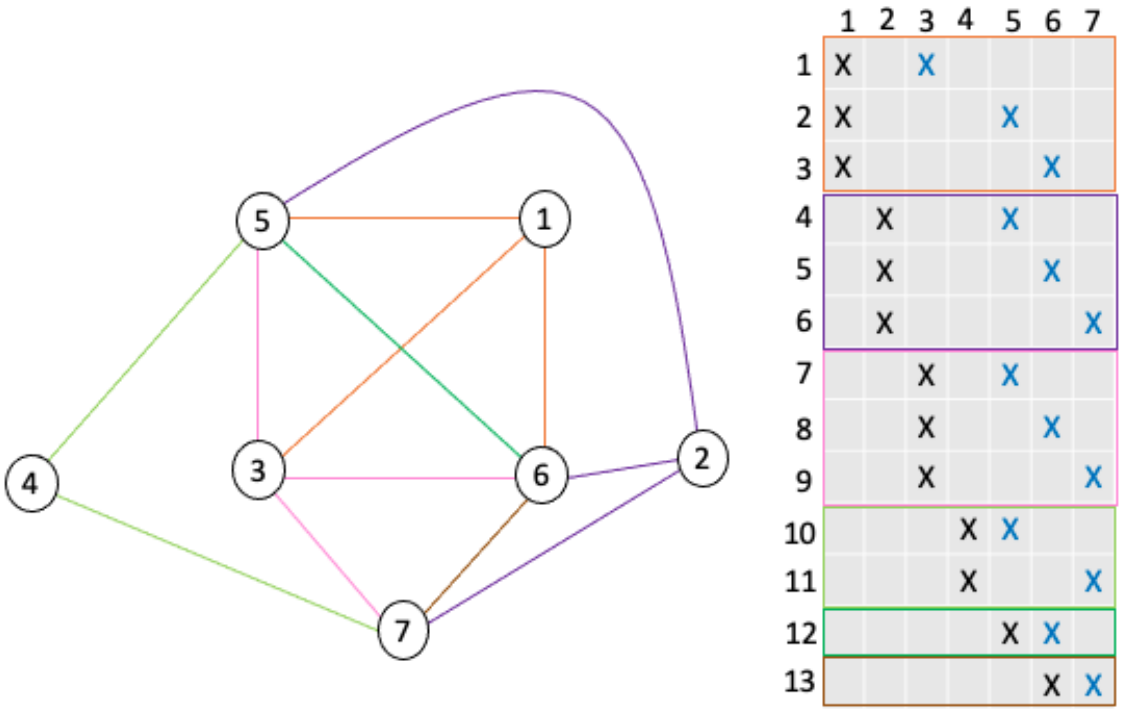


Figure 5.2: Intersection matrix representation of the example input graph

5.2.1 Adjacency Matrix-based Triangle Counting

Many of the existing triangle counting methods use the sparse representation of adjacency matrices in their calculations. The adjacency matrix $A(G) \equiv A \in \{0,1\}^{|V|\times|V|}$ associated with graph G is defined as,

$$A(i,j) = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E, i \neq j \\ 0 & \text{otherwise} \end{cases}$$

It is well known in the literature that the number of closed walks of length $k \geq 0$ are obtained in the diagonal entries of k^{th} power A^k. Therefore, the total number of triangles in a graph G, $\Delta(G)$, is given by the trace of A^3,

$$\Delta(G) = \frac{1}{6} Tr(A^3).$$

The factor of $\frac{1}{6}$ accounts for the multiple counting of a triangle (the number of ways closed

walks of length 3 can be obtained is $3! = 6$). There is a large body of literature on sparse linear algebraic triangle counting methods based on adjacency matrix representation of the data [8]. In [83], the miniTri's triangle counting implementation takes the adjacency matrix A of the input graph and creates an incidence matrix B. Then the enumeration and counting of the triangles occur in the overloaded matrix multiplication [83]. Here, an entry in the resultant matrix with a value of 2 corresponds to a completed triangle. miniTri method triple-counts each triangle, once for each vertex, so the final result is divided by 3 to give the total number of triangles in the graph. Since the multiplication of two sparse matrices usually results in a dense matrix, this is a memory intensive process.

5.2.2 Intersection Matrix-based Triangle Counting

Graph algorithms can be effectively expressed in terms of linear algebra operations [46], and we combine this knowledge with our proposed data representation to count the triangles in a structured three-step method. For vertex i we first find its neighbors $j > i$ such that $\{i, j\} \in E$ by multiplying the submatrix of X consisting of rows corresponding to edges incident on i (let us call them $(i-j)-$rows) by the transpose of the vector of ones of compatible length. A value of 1 in the vector-matrix product indicates that the corresponding vertex j is a neighbor of vertex i.

Next, we multiply the submatrix of X consisting of columns j identified in the previous step and the rows below the $(i-j)-$rows by a vector of ones of compatible length. A value of 2 in the matrix-vector product indicates a triangle of the form (i, j, j') where j and j' are neighbors of vertex i with $j < j'$. Let k be the row index in matrix X for which the matrix-vector product contains a 2. Then it must be that $X(k, j) = 1$ and $X(k, j') = 1$. Since each row of X contains exactly 2 nonzero entries that are 1, it follows that $\{j, j'\} \in E$. The forward neighbor of a vertex v is the set of neighbors of v having a higher label than v. The mentioned operation to get the matrix-vector product is identical to performing a set intersection on the forward neighbors of vertices j and j'.

The number of triangles in the graph is given by the sum of the number of triangles associated with each vertex as described. Since the edges are represented in sorted order in our algorithm, unlike many other triangle counting methods [83], each triangle is counted exactly once. Figure 5.2 displays the intersection matrix representation of the input graph X. The triangles of the form $(1, j, j')$ where $j, j' \in \{3,5,6\}$ are obtained from the product $X(7:13,[3\ 5\ 6]) * \mathbf{1}$, where $\mathbf{1}$ denotes the vector of ones. The product has a 2 at locations corresponding to rows 7, 8, and 12 of X and the associated triangles are $(1,3,5),(1,3,6)$, and $(1,5,6)$. Therefore, there are three triangles incident on vertex 1, and we can easily verify the graph contains a total of 7 triangles across all of the vertices.

5.2.3 Data Structure

In our preliminary implementation, we use two arrays to store useful information that can be computed after we sort the edges. `FDC` (Forward Degree Cumulative) is an array of size n, with elements corresponding to the total number of "forward neighbors" across the vertices of a graph. Forward neighbors are defined as the neighbors of a vertex that have a higher label than the vertex of interest. With the vertices of the graph labelled, finding the forward degree of a vertex j can be calculated as `fd(j) = FDC[j+1] - FDC[j]`. `FN` is an array of size m that stores *which* vertices are the forward neighbors of a vertex j. Using `FN` we can find these forward neighbors of j as `fn(j) = FN[k]`, where k ranges from `FDC[j]` to `FDC[j+1]-1`. The arrays `FDC` and `FN` thus save the vector-matrix products needed to find the forward neighbors. Figure 5.3 displays the arrays `FDC` and `FN` for the graph of Figure 5.2.

FN =	3	5	6	5	6	7	5	6	7	5	7	6	7
FDC =	1	4		7		10		12		13		14	

Figure 5.3: FN and FDC for the example graph

To identify a triangle we need to identify it's three corner points (vertices). For a vertex

j, we can get its forward neighbors from `FN` array. Then we need to identify the common forward neigbors between j and forward neighbors of j, `fn(j) = FN[k]`, where k ranges from `FDC[j]` to `FDC[j+1]-1`. We have implemented three different techniques to get these common forward neighbors between j and $fn(j)$.

- **Version 1.** The first version is using `std::set_intersection`, the standard set intersection operation. This operation constructs a sorted range beginning in the location pointed by the result with the set intersection of the two sorted ranges. This operation has linear time complexity concerning the sizes of the lists.
- **Version 2.** In version 2, we avoid using standard set intersection operation and implement our intersection operation to get the common forward neighbors between j and $fn(j)$. The complexity is the same as standard intersection operation.
- **Version 3.** In this third version, our computer implementation uses a sparse matrix framework of DSJM [32] and expresses all computations in terms of intersection matrices.

5.2.4 Local Triangle Count and Edge Support

As discussed in Section 5.1, there are many other metrics related to triangle computation that can be found using our intersection matrix data structure. The bases for these metrics are the triangle degrees, which are the number of triangles incident on an edge (edge support) or vertex (local triangle count) of a graph. This is illustrated in Figure 5.4 as **edgeDeg** and **vertDeg**, respectively, derived from Figure 5.1.

edgeDeg =	2	2	2	1	2	1	2	3	1	0	0	3	2
vertDeg =	3		2		4		0		4		6		2

Figure 5.4: vertDeg and edgeDeg for the example graph

Let j be the column (vertex) of matrix X (graph G) currently being processed in the **fullCount** algorithm. For each pair of its forward neighbors j' and j'' there is an edge

between them if and only if both of the corresponding columns contain a 1 in some row k identifying the triangle (j, j', j''). In terms of the matrix-vector multiplication in line 7 of algorithm **fullCount**, vector T will get updated as $T(k) \leftarrow 2$. Thus the triangle (j, j', j'') can be enumerated and stored instantly. The vertex triangle degrees of each triangle are dynamically updated with this same information, and stored in an array. The edge triangle degrees are stored in a separate array and updated by exploiting the structure of the `FN` and `FDC` arrays in tandem. The entries of the `FDC` array, while primarily used to store the forward degree of a vertex, also contain the edge number (edge id) that the forward neighborhood of the vertex of interest begins and ends at. Since the sub-arrays in `FN` that correspond to the forward neighborhood of the vertices are in the same order as the listed edges of the intersection matrix, any edge between two vertices can be identified by first finding the distance between the higher labelled vertex and the beginning of the forward neighborhood in which it is found (using `FN`), and then adding this distance to the entry in `FDC` that corresponds to the edge of the lower numbered vertex. Finally, the k-count distribution of the triangles is used to give a bound on the maximum clique of a graph [83], and with the triangles enumerated and the edge and vertex triangle degrees computed and stored as shown in Figure 5.4, the k-count calculations can be quickly computed using the method described Section 5.1.

Triangle centrality helps find important vertices in a graph basing the concentration of triangles surrounding each vertex. With the edge and vertex triangle degrees computed and stored as shown in Figure 5.4, the triangle centrality calculations can be computed using the method described in Section 5.1. For this calculation, we do not require storing the enumerated triangles.

5.2.5 Algorithm

The algorithm in its entirety is given in this section (as Algorithm 5), and for calculating the triangle centrality, we can avoid lines 14 and 15.

Algorithm 5 fullCount (X)

```
   Input: Intersection matrix X
 1 Calculate FDC                                   ▷ Forward degree cumulative
 2 Calculate FN                                    ▷ Forward neighbor
 3 count ← 0                                       ▷ Number of triangles
 4 for j = 1 to n − 1 do                           ▷ j ∈ V, where V is the set of vertices
 5     fd ← FDC[j+1] − FDC[j]                      ▷ fd is the forward degree of j
 6     if fd > 1 then                              ▷ j has more than one forward neighbor
 7         T = X([FDC(j+1) : m], fn_j) ∗ 1
 8         S = {t | T[t] = 2}
 9         if S ≠ ∅ then
10             count ← count + |S|
11             for t ∈ S do
12                 update edgeDeg                  ▷ Array of triangle edge degrees
13                 update vertDeg                  ▷ Array of triangle vertex degrees
14                 Triangles ← Triangles ∪ t       ▷ Array that stores enumerated triangles
15 kCountTable ← computeKCounts(count, vertDeg, edgeDeg, Triangles)
16 return count, vertDeg, edgeDeg, kCountTable, and Triangles
```

5.3 Numerical Results

This section contains experimental results from selected test instances. The first set comprises real-world social networks from the Stanford Network Analysis Project (SNAP), obtained from the Graph Challenge website [72]. SNAP is a collection of more than 50 large network datasets containing a large number of nodes and edges, including social networks, web graphs, road networks, internet networks, citation networks, collaboration networks, and communication networks [49]. The first set of experiments were performed using a PC with a 4th Gen Intel Core I7-4770 Processor (Quad Core HT, with 3.4GHz Turbo and 8GB RAM), running Centos Linux v7.9. The implementation language was *C++* and the code was compiled using $-O3$ optimization flag with a *g++* version 4.4.7 compiler. Performance times are reported in seconds and were averaged over three runs where possible, using the following implementation abbreviations: *mt1* for *miniTri1*, *mt2* for *miniTri2*, and *int* for our intersection algorithm.

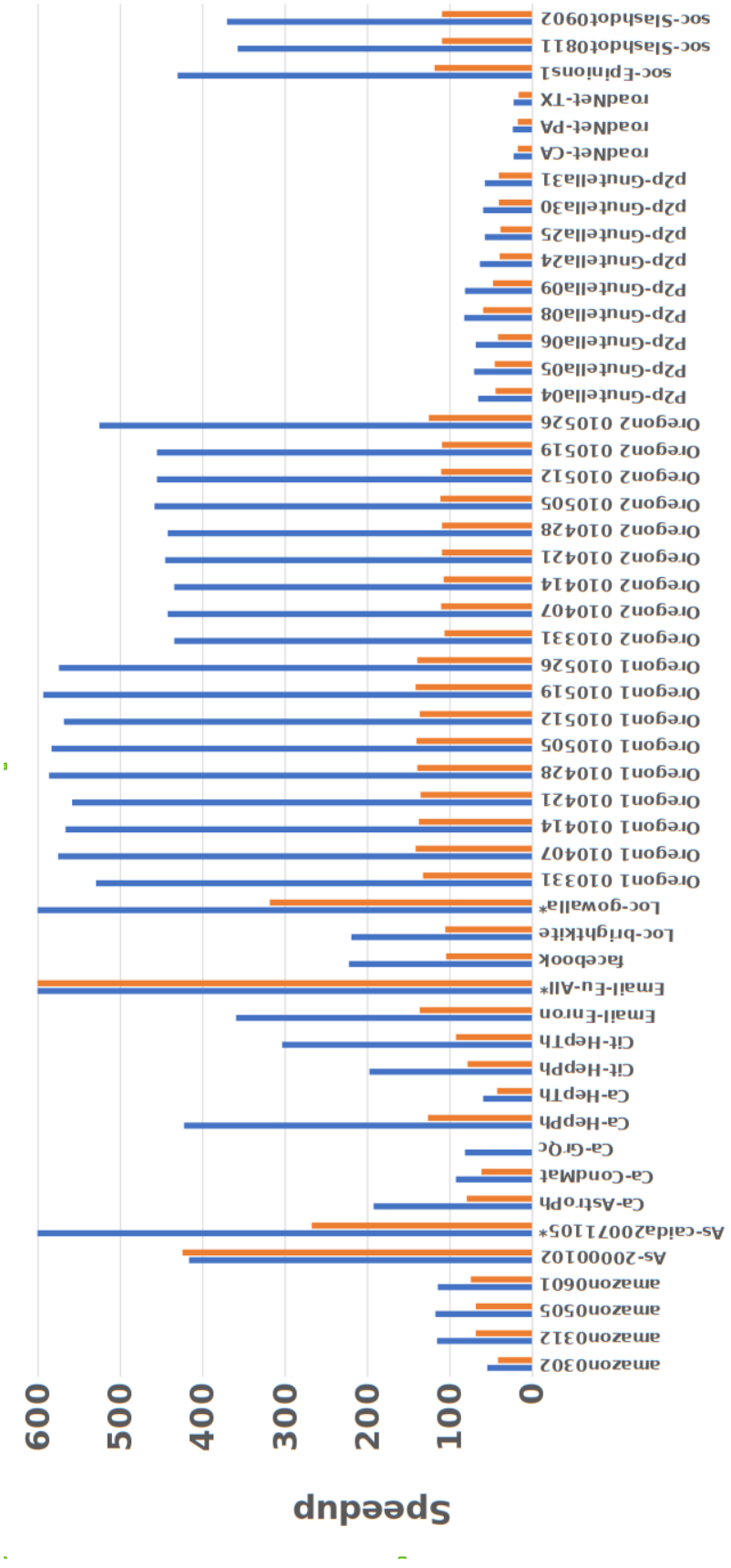


Figure 5.5: Comparing our intersection algorithm with both miniTri implementations on large real-world networks

5.3.1 Traingle Counting Algorithm

Figure 5.5 shows the speedups of our algorithm versus the two reference miniTri implementations on these real-world instances. The speedups ranged from 22× to an impressive 1383× over *miniTri1*, and from 16× to 642× over *miniTri2*, with two instances ("flickrEdges" and "Cit-Patents") failing to compute with *miniTri2*. Instances with an * had speedups greater than 650× against *miniTri1* and were cut off from the figure for ease of viewing.

Table 5.1: Comparing Our Intersection Algorithm with miniTri on Large Synthetic Networks

Graph Characteristics				Time in Seconds	
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	mt1	int
graph500-scale18-ef16	262144	4194304	82287285	17440	9.357
graph500-scale19-ef16	524288	8388608	186288972	49211.8	25.21
graph500-scale20-ef16	1048576	16777216	419349784	197456	72.34
graph500-scale21-ef16	2097152	33554432	935100883	N/A	171.2
graph500-scale22-ef16	4194304	67108864	2067392370	N/A	481.43
graph500-scale23-ef16	8388608	134217728	4549133002	N/A	1340.05
graph500-scale24-ef16	16777216	268435456	9936161560	N/A	3317.15
graph500-scale25-ef16	33554432	536870912	21575375802	N/A	7959.39

Table 5.1 compares our algorithm performance on large synthetic test instances from GraphChallenge to *miniTri1* (*miniTri2* was only able to compute the first instance and thus omitted). "N/A" denotes instances where *miniTri1* timed out after four days of computation. Due to the large sizes of this second set of instances, they were run on the large High Performance Computing system (Graham cluster) at Compute Canada. On the first 3 instances, our method is over 1800 times faster than *miniTri1*, and the relative performance improves with increasing instance size, further demonstrating the scalability of our triangle counting algorithm.

Figure 5.6 demonstrates our algorithm's performance on relatively dense brain networks from the Network Repository [70], back in the Linux environment. These graphs have

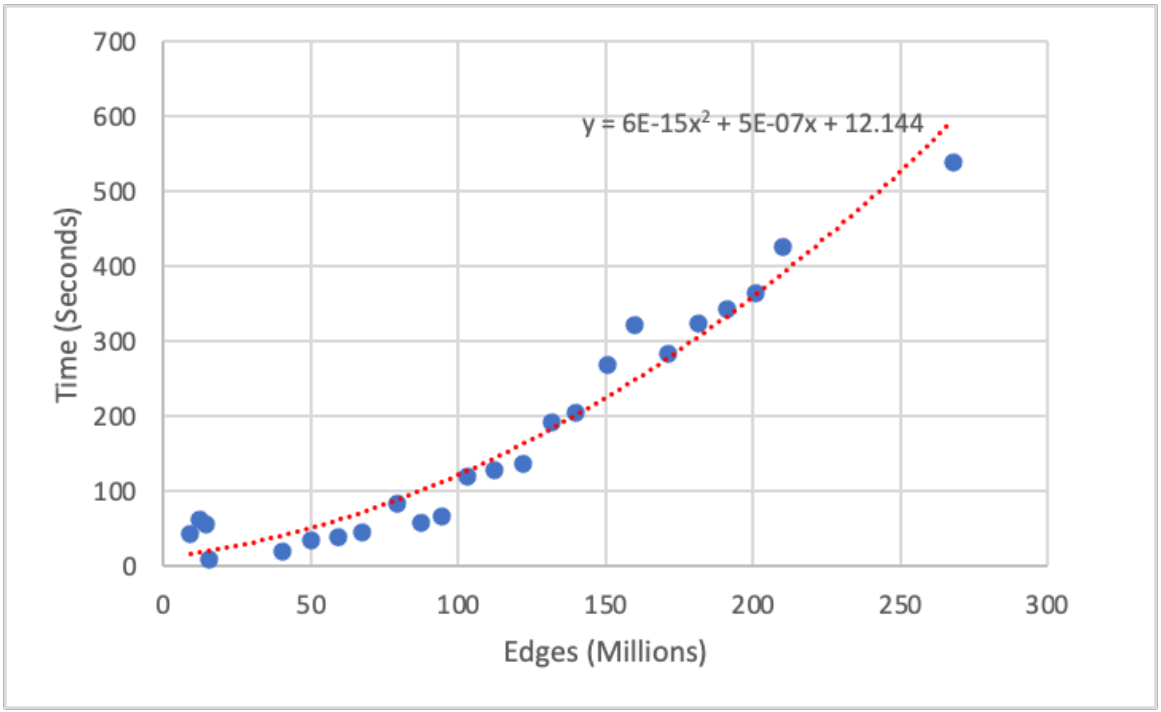


Figure 5.6: Testing our intersection algorithm on networks with billions of triangles

between 15 and 268 million edges and up to 42 trillion triangles, and neither miniTri implementation was able to provide results for any of the instances. The line of best fit is $(y = 6 \times 10^{-15}x^2 + 5 \times 10^{-07}x + 12.144)$, a polynomial of degree 2 and shows that our algorithm scales very well with graphs with massive amounts of triangles.

Our intersection-based implementation also produces competitive results when compared to the state-of-the art triangle counting algorithms [73]. Algorithms were analyzed and compared by fitting a model of graph counting times, T_{tri}, as a function of the number of edges $N_e = |E|$. This data was then used to estimate the parameters N_1 (the number of edges that can be processed in one second) and β:

$$T_{tri} = (N_e/N_1)^{\beta}$$

to compare different counting implementations. Implementations with a larger N_1 and smaller β perform the best, and the top entries from the 2019 review had N_1 values ranging from 5×10^5 to 5×10^8, and β values ranging from $\frac{1}{2}$ to $\frac{4}{3}$ [73]. For reference, our algorithm

had $\beta = \frac{3}{4}$ and $N_1 = 1 \times 10^7$.

5.3.2 Triangle Counting, Triangle Vertex & Edge degree, and k-Count Calculations

After examining the comparative performance of our triangle counting algorithm, we proceeded to expand the implementation to include the metrics described in Section 5.2.4 - triangle counting, triangle vertex degree, triangle edge degree, and k-count calculations. Similar to the basic counting experimental results, our intersection method of this "full count" was faster than *miniTri1* and *miniTri2* on every instance. *miniTri1*, *miniTri2*, and our full count intersection algorithm were run on the large High Performance Computing system (Graham cluster) at Compute Canada. The speedups range between $3\times$ and $839\times$ on fifty-three instances, displayed in Table 5.2. One noteworthy observation about these results is that due to the data structure that stored the enumerated triangles, the k-count calculation of our algorithm ran much faster than those of miniTri, even though the code implementation was nearly identical. This demonstrates the versatility of *FDC* and *FN* in their ability to perform a wide range of network analytics.

Table 5.2: Comparing Our Full Count Intersection Algorithm with *miniTri1* and *miniTri2* on Large Real World Networks

Graph Characteristics				Time in Seconds			Speedup	
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	mt1	mt2	int	mt1/int	mt2/int
amazon0302	262111	899792	717719	10.2	3.34	0.38	26.84	8.79
amazon0312	400727	2349869	3686467	94.57	21.06	1.87	50.57	11.26
amazon0505	410236	3356824	3951063	102.76	22.26	1.95	52.70	11.42
amazon0601	403394	3387388	3986507	98.21	24.4	1.99	49.35	12.26
As-20000102	6474	25144	6584	1.07	0.88	0.01	107.00	88.00
As-caida20071105	31379	53381	36365	9.81	2.66	0.03	327.00	88.67
Ca-AstroPh	18722	198050	1351441	17.75	4.32	0.55	32.27	7.85
Ca-CondMat	23133	93439	173361	2.22	0.68	0.07	31.71	9.71
Ca-GrQc	5242	14484	48260	0.17	0.08	0.02	8.50	4.00
Ca-HepPh	12008	118489	3358499	28.23	5.32	1.62	17.43	3.28
Ca-HepTh	9877	25973	28339	0.22	0.1	0.01	22.00	10.00

Table 5.2 – *Continued on next page*

Table 5.2 – *Continued from previous page*

Graph Characteristics				Time in Seconds			Speedup	
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	mt1	mt2	int	mt1/int	mt2/int
Cit-HepPh	32233	417294	1276868	38.4	8.2	0.6	64.00	13.67
Cit-Patents	3774768	33037894	7515023	421.65	115.4	8.75	48.19	13.19
Cit-HepTh	24982	348238	1478735	58.1	11.17	0.68	85.44	16.43
Email-Enron	36692	183830	727044	51.58	8.21	0.37	139.41	22.19
Email-Eu-All	265214	364481	267313	209.73	52.51	0.25	838.92	210.04
facebook	4039	88234	1612010	13	3.29	0.66	19.70	4.98
flickrEdges	105938	2316948	107987357	2808.79	277.47	89.58	31.36	3.10
Loc-brightkite	58228	214078	494728	22.1	4.51	0.24	92.08	18.79
Loc-gowalla	196591	950327	2273138	658.74	74.29	1.7	387.49	43.70
Oregon1 010331	11492	22002	17144	3.44	0.65	0.01	344.00	65.00
Oregon1 010407	11492	21999	15834	3.42	0.65	0.01	342.00	65.00
Oregon1 010414	11492	22469	18237	3.56	0.68	0.01	356.00	68.00
Oregon1 010421	11492	22747	19108	3.6	0.69	0.01	360.00	69.00
Oregon1 010428	11492	22493	17645	3.84	0.69	0.01	384.00	69.00
Oregon1 010505	11492	22607	17597	3.9	0.69	0.01	390.00	69.00
Oregon1 010512	11492	22677	17598	3.64	0.69	0.01	364.00	69.00
Oregon1 010519	11492	22724	17677	3.71	0.71	0.01	371.00	71.00
Oregon1 010526	11492	23409	19894	4.12	0.73	0.01	412.00	73.00
Oregon2 010331	11806	31180	82856	4.47	0.84	0.04	111.75	21.00
Oregon2 010407	11806	30855	78138	4.56	0.85	0.04	114.00	21.25
Oregon2 010414	11806	31761	88905	4.69	0.88	0.04	117.25	22.00
Oregon2 010421	11806	31538	82129	4.77	0.88	0.04	119.25	22.00
Oregon2 010428	11806	31434	78000	4.81	0.88	0.04	120.25	22.00
Oregon2 010505	11806	30943	72182	4.63	0.86	0.04	115.75	21.50
Oregon2 010512	11806	31303	72866	4.72	0.87	0.04	118.00	21.75
Oregon2 010519	11806	32287	83709	5.2	0.91	0.04	130.00	22.75
Oregon2 010526	11806	32730	89451	5.27	0.94	0.04	131.75	23.50
P2p-Gnutella04	10879	39994	934	0.29	0.16	0.005	58.00	32.00
P2p-Gnutella05	8846	63678	1112	0.25	0.13	0.004	62.50	32.50
P2p-Gnutella06	8717	63050	1142	0.25	0.13	0.004	62.50	32.50
P2p-Gnutella08	6301	41554	2383	0.2	0.12	0.003	66.67	40.00
P2p-Gnutella09	8114	52026	2354	0.24	0.12	0.004	60.00	30.00
p2p-Gnutella24	26518	65369	986	0.43	0.21	0.006	71.67	35.00

Table 5.2 – *Continued on next page*

Table 5.2 – *Continued from previous page*

Graph Characteristics				Time in Seconds			Speedup	
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	mt1	mt2	int	mt1/int	mt2/int
p2p-Gnutella25	22687	54705	806	0.32	0.16	0.005	64.00	32.00
p2p-Gnutella30	36682	88328	1590	0.56	0.31	0.009	62.22	34.44
p2p-Gnutella31	62586	147891	2024	0.98	0.48	0.02	49.00	24.00
roadNet-CA	1965206	5533214	120676	6.55	2.26	0.22	29.77	10.27
roadNet-PA	1090920	1541898	67150	3.82	1.3	0.12	31.83	10.83
roadNet-TX	1393383	1921660	82869	4.52	1.62	0.15	30.13	10.80
soc-Epinions1	75888	405740	1624481	178.97	25.19	1.08	165.71	23.32
soc-Slashdot0811	77360	469180	551724	152.31	18.71	0.49	310.84	38.18
soc-Slashdot0902	82168	504230	602592	167.39	20.27	0.52	321.90	38.98

5.3.3 Triangle Centrality and Ranking

We refer our intersection algorithm as **fullCount** when it calculates triangles, triangle vertex degree, triangle edge degree, and *k*-count. The dense brain networks from the Network Repository [70] and the large synthetic test instances from GraphChallenge require huge memory space to store the enumerated triangles. Therefore, to test the scalability and runtime of these dense graphs, we avoid calculating *k*-count and only calculate triangles, triangle vertex degree, and triangle edge degree. These metrics are essential to find triangle centrality and rank the vertices. *miniTri1* and *miniTri2* failed to calculate the number of triangles for these dense graphs. Therefore, we report the runtime of our intersection algorithm without *k*-count. Due to the large sizes of these dense instances, they were run on the High-Performance Computing system (Graham cluster) at Compute Canada. Table 5.3 reports our algorithm's performance on brain networks and Table 5.4 reports the running time required for the synthetic test instances from GraphChallenge, `Graph-500`.

Li and Bader [51] presented a rapid implementation of triangle centrality using SuiteSparse GraphBLAS (Version 5.1.5) [45] an API specification for describing graph algorithms in the language of linear algebra, and the implementation language was *C*. We refer to their implementation method as `TC-GrB`. We use our `int` algorithm to calculate triangle central-

Table 5.3: Test Results of Our Intersection Algorithm without k-count on Dense Brain Networks

Graph Characteristics				Time in Seconds
Name (bn-human)	$\|V\|$	$\|E\|$	$\Delta(G)$	int
BNU_1_0025890_session_1	177,584	15,669,037	662694994	324.928
Jung2015_M87125334	763,149	40,258,003	1515479025	360.075
Jung2015_M87104509	737,579	50,037,313	2512591873	853.852
Jung2015_M87119472	835,832	59,548,327	3023865951	932.158
Jung2015_M87104300	851,113	67,658,067	3516573387	1208.99
Jung2015_M87118347	428,842	79,114,771	6884218472	3308.74
Jung2015_M87102575	935,265	87,273,967	4732564614	1660.09
Jung2015_M87122310	924,284	94,370,886	5577667716	1895.23
BNU_1_0025914_session_2	701,145	103,134,404	9531928703	4003.07
BNU_1_0025916_session_1	714,571	112,519,748	10147347192	4135.25
Jung2015_M87118219	791,219	121,907,663	11287089562	4688.94
BNU_1_0025889_session_2	742,862	131,926,773	14550152774	7473.56
BNU_1_0025873_session_2-bg	692,397	140,102,158	16480195100	8636.88
BNU_1_0025868_session_1	727,487	150,443,553	18944842260	12778.2
BNU_1_0025918_session_1	748,521	159,835,566	23287278951	18409.3
Jung2015_M87112427	728,874	171,231,873	21603267930	11800.3
Jung2015_M87118465	774,886	181,569,095	24857761263	17265.8
Jung2015_M87104201	707,284	191,224,983	25942442887	16681.4
Jung2015_M87101705	776,644	201,198,184	26497580631	16450.6
Jung2015_M87125286	753,905	209,976,387	34150382906	27672.8
Jung2015_M87113878	784,262	267,844,669	41727013307	28300.8

Table 5.4: Test Results of Our Intersection Algorithm without k-count on Large Synthetic Instances from GraphChallenge

Graph Characteristics				Time in Seconds
Name (graph500)	$\|V\|$	$\|E\|$	$\Delta(G)$	int
scale18-ef16	262144	4194304	82287285	107.739
scale19-ef16	524288	8388608	186288972	352.135
scale20-ef16	1048576	16777216	419349784	1104.34
scale21-ef16	2097152	33554432	935100883	3217.23
scale22-ef16	4194304	67108864	2067392370	10565.3
scale23-ef16	8388608	134217728	4549133002	33227
scale24-ef16	16777216	268435456	9936161560	97678.2
scale25-ef16	33554432	536870912	21575375802	311789

Table 5.5: Performance Comparison between GraphBLAS and Our `int` Algorithm for Implementing Triangle Centrality

Graph Characteristics				Time in Seconds			
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	TC-GrB	Our Method		
					int	TC	Total
Com-Youtube	1134890	2987624	3056386	3.84	3.41	0.02	**3.43**
as-Skitter	1696415	11095298	28769868	**9.66**	60.97	0.05	61.02
com-LiveJournal	3997962	34681189	177820130	47.5	43.25	0.28	**43.53**
com-Orkut	3072441	117185083	627584181	642.74	468.01	1.04	**469.05**

ity. First, we calculate the total and local triangle count. Then using the triangle centrality equation, we calculate the triangle centrality. We report runtime for triangle count, triangle vertex & edge degree calculation separately from triangle centrality calculation. Our implementation language was *C++* and the code was compiled with a *g++* version 4.4.7 compiler in ASUS VivoBook Flip 14 PC with a 7th Gen Intel Core I5-7200U Processor (Dual Core, with 2.5GHz and 5GB RAM), running Linux Mint 19. For a fair comparison, we test `TC-GrB` in the same PC.

Table 5.5 shows the performance comparison between GraphBLAS and our `int` algorithm for implementing triangle centrality. Our method can calculate triangle centrality within a fraction of a second for all the instances. Our triangle centrality calculation re-

quires triangle counting and triangle vertex & edge degree calculation. Therefore, if we consider the running time for our `int` algorithm with our triangle centrality calculation, still we get better performance except for one instance `as-Skitter`.

Triangle centrality depends on the number of triangles and the triangle vertex & edge degrees. Therefore, finding important vertices from the given graph is one of the important application of our `int` algorithm. Burkhardt [9] introduced triangle centrality for identifying important vertices in a graph. The author also presented a parallel version of his method. We will discuss the parallel version of our algorithm in the next section. However, we get the same highest-ranked vertex for graphs `Borgatti 2006 Figure 3`, `Zachary's Karate club`, and `Lusseau's Dolphin network` as Burkhardt. Detail of these graphs can be found in [9].

5.4 Parallel Algorithms

In this section, we describe the parallel implementation of our intersection algorithm, **fullCount**. We target to parallelize our basic triangle counting algorithm. Then we discuss the parallel algorithm for our **fullCount** algorithm concerning triangle count, local triangle count, and k-count. Finally, we present a parallel version of our triangle centrality calculation method.

Burkhardt [9] presented a Parallel Random Access (PRAM) algorithms for triangle centrality. Burkhardt mentioned Concurrent Read Concurrent Write (CRCW) PRAM permits concurrent read and write to a memory location by any number of processors. However, a resolution protocol handles values concurrently written by multiple processors to the same memory location. Instead of following the same steps as Burkhardt, we are interested in analyzing the speed-up we achieve from our parallel algorithms.

We run our parallel algorithms on the High-Performance Computing system (Graham cluster) at Compute Canada. The implementation language was *C++*, and we varied the number of threads between 1 and 16 while using OpenMP.

OpenMP (Open Multi-Processing) is an application programming interface (API) for shared memory parallel computing. It is supported on numerous platforms, including Linux and Windows, and is available for the C/C++ and Fortran programming languages. The API consists of directives, a software library, and environment variables.

5.4.1 Basic Triangle Counting Algorithm

We describe our parallel triangle count algorithm in this section. In Section 5.2.2, we discussed in detail the serial version of this triangle count algorithm. Algorithm 6 presents the complete parallel triangle count algorithm. In the parallel region, all the threads count triangle using the private variable, *local_count*. After all the threads complete counting triangles locally, the master thread then updates the *count* variable in the critical region to avoid the race condition.

Algorithm 6 ParallelBasicCount (X)

```
   Input: Intersection matrix X
 1 Calculate FDC                                          ▷ Forward degree cumulative
 2 Calculate FN                                           ▷ Forward neighbor
 3 count ← 0                                              ▷ Number of triangles
 4 parallel region
 5     local_count ← 0                                    ▷ Local number of triangles
 6     do in parallel
 7         for j = 1 to n − 1 do                          ▷ j ∈ V, where V is the set of vertices
 8             fd ← FDC[j+1] − FDC[j]                     ▷ fd is the forward degree of j
 9             if fd > 1 then                             ▷ j has more than one forward neighbor
10                 T = X([FDC(j+1) : m], fn_j) ∗ 1
11                 S = {t | T[t] = 2}
12                 if S ≠ ∅ then
13                     local_count ← local_count + |S|
14     critical region
15         count ← count + local_count
16 return count
```

The brain networks from the Network Repository [70] are relatively dense. These graphs have between 15 and 268 million edges and up to 42 trillion triangles. `Graph-500` network is the large synthetic test instances from GraphChallenge. Our serial triangle count

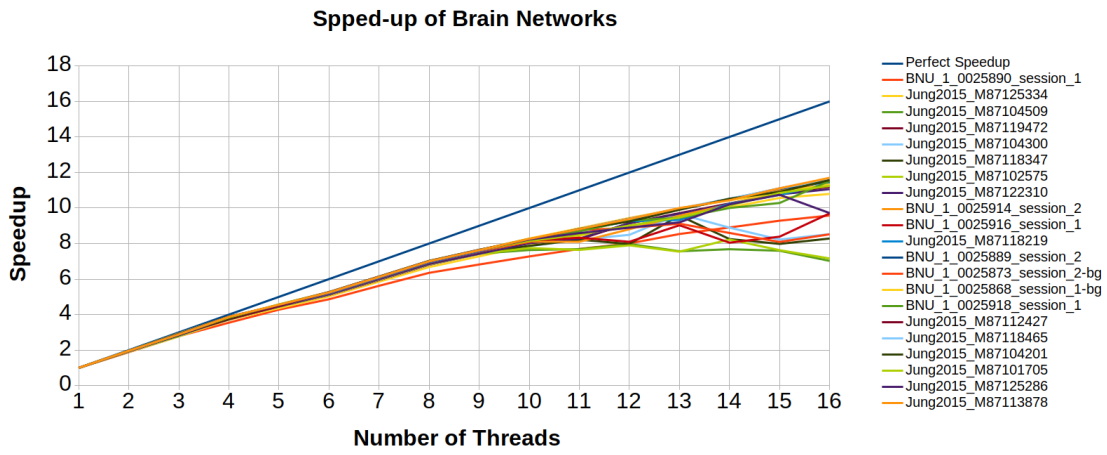


Figure 5.7: Speed-up of brain networks for parallel basic count algorithm

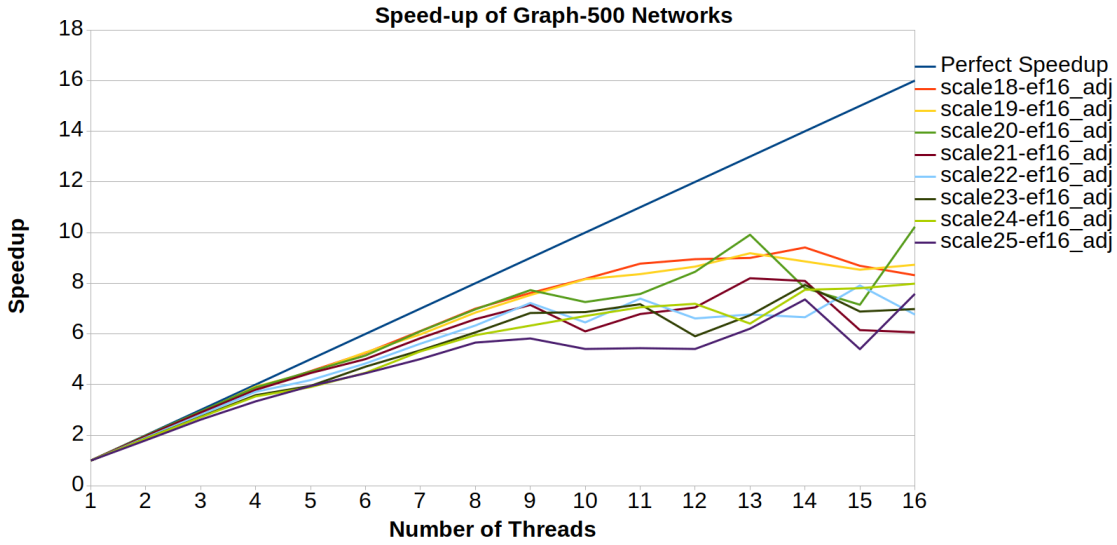


Figure 5.8: Speed-up of Graph-500 networks for parallel basic count algorithm

algorithm showed that our algorithm scales very well with graphs with massive amounts of triangles where the existing miniTri implementations could not provide results for any of the instances. Therefore, to test our parallel implementation's scalability, we are interested in testing for these instances. Figure 5.7 shows the speed-up of brain networks, and Figure 5.8 presents the speed-up of Graph-500 networks. Our parallel basic triangle count algorithm shows perfect linear speed-up up to four threads and close to perfect linear speed-up up to eight threads.

5.4.2 fullCount Algorithm

We describe our parallel **fullCount** algorithm in this section. We described in detail the serial version of this algorithm in Section 5.2.4. Algorithm 7 presents the complete parallel **fullCount** algorithm. The k-count is one of the applications of triangle count and enumeration. We compute k-count after we enumerate triangles. Algorithm 7 includes parallel k-count along with triangle count, edge support, and local triangle count.

Each thread counts triangles using the private variable, *local_count* in the parallel region. There are other metrics too. *local_edgeDeg* tracks the number of triangle incidents on edge (edge support), and *local_vertDeg* tracks the number of triangles incident on the vertex (local triangle count)—finally, *loacal_Triangles* stores the enumerated triangles for the thread. After all the threads complete their task locally, the master thread updates all the variables in the critical region to avoid the race condition.

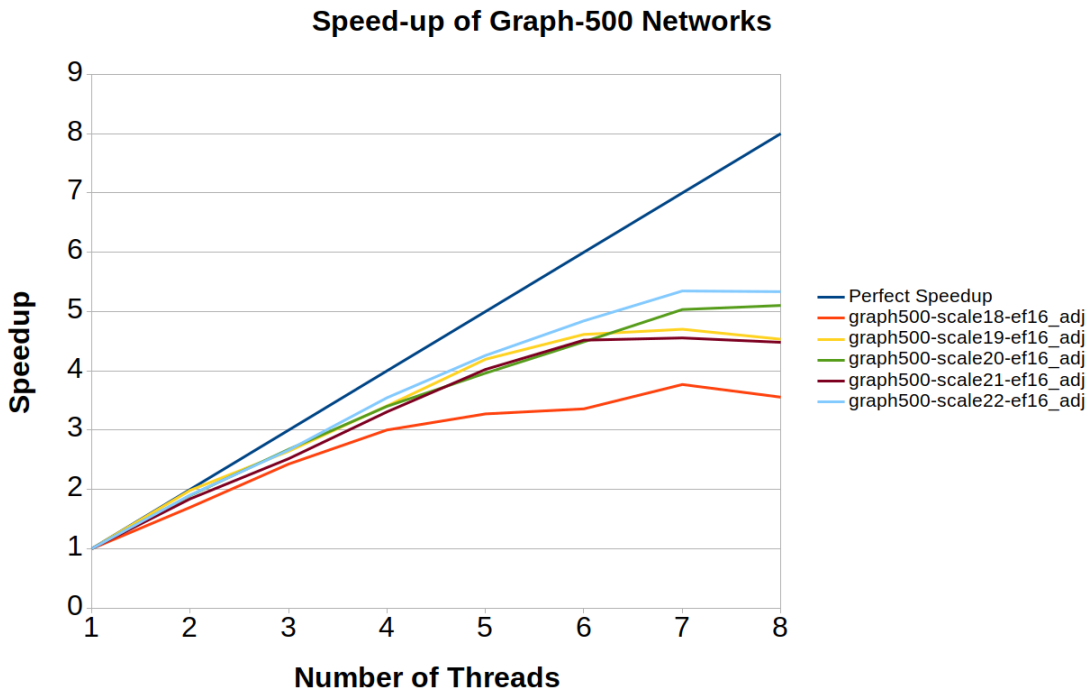


Figure 5.9: Speed-up of selected test instances for **ParallelFullCount** algorithm

Figure 5.9 presents the speed-up of Graph-500 instances for **ParallelFullCount** algorithm including the k-count. Up to using four threads, we observe speedups close to the

Algorithm 7 ParallelFullCount (X)

```
    Input: Intersection matrix X
 1  Calculate FDC                                          ▷ Forward degree cumulative
 2  Calculate FN                                           ▷ Forward neighbor
 3  count ← 0                                              ▷ Number of triangles
 4  parallel region
 5      local_count ← 0                                    ▷ Local number of triangles
 6      do in parallel
 7          for j = 1 to n − 1 do                          ▷ j ∈ V, where V is the set of vertices
 8              fd ← FDC[j + 1] − FDC[j]                   ▷ fd is the forward degree of j
 9              if fd > 1 then                             ▷ j has more than one forward neighbor
10                  T = X([FDC(j + 1) : m], fn_j) ∗ 1
11                  S = {t | T[t] = 2}
12                  if S ≠ ∅ then
13                      local_count ← local_count + |S|
14                      for t ∈ S do
15                          update local_edgeDeg           ▷ Local array of triangle edge degrees
16                          update local_vertDeg           ▷ Local array of triangle vertex degrees
17                          local_Triangles ← local_Triangles ∪ t   ▷ Local array storing
    enumerated triangles
18      critical region
19          count ← count + local_count
20          for j = 1 to m do
21              edgeDeg[j] ← edgeDeg[j] + local_edgeDeg[j]
22              if j ≤ n then
23                  vertDeg[j] ← vertDeg[j] + local_vertDeg[j]
24          Triangles ← Triangles ∪ local_Triangles
25  kCountTable ← ParallelComputeKCounts(count, vertDeg, edgeDeg, Triangles)
26  return count, vertDeg, edgeDeg, kCountTable, and Triangles
```

perfect speedup line, and beyond eight threads, we do not see any speedup. The reason is storing and accessing an extensive amount of triangles for these dense graphs.

5.4.3 Triangle Centrality

Triangle centrality helps find important vertices in a graph basing the concentration of triangles surrounding each vertex. An important vertex in triangle centrality is at the center of many triangles. Such vertex may present in many triangles. To find the triangle centrality of each vertex, we do not require storing the enumerated triangles. So we avoid lines 24, and 25 of Algorithm 7 for the **ParallelFullCount** and then use Algorithm 8 to calculate the triangle centrality.

The **ParallelTriangleCentrality** algorithm is straightforward and has no chance of race conditions. Therefore, we calculate triangle centrality for each vertex $j \in V$ using a parallel for loop.

Algorithm 8 ParallelTriangleCentrality (X, $vertDeg$, $Total\_Triangles$)

Input: Intersection matrix X, $vertDeg$, Totoal triangle count $Total\_Triangles$

1 **do in parallel**
2 **for** $j = 1$ to n **do** ▷ $j \in V$, where V is the set of vertices
3 $core\_triangle\_sum \leftarrow vertDeg[j]$
4 $non\_core\_triangle\_sum \leftarrow 0$
5 **for** $k \in neighbors(j)$ **do** ▷ Find $neighbors(j)$ using sparse matrix data structure
6 **if** k forms a triangle with j **then**
7 $core\_triangle\_sum \leftarrow core\_triangle\_sum + vertDeg[k]$
8 **else**
9 $non\_core\_triangle\_sum \leftarrow non\_core\_triangle\_sum + vertDeg[k]$
10 $TC[j] \leftarrow \frac{\frac{1}{3} \times core\_triangle\_sum + non\_core\_triangle\_sum}{Total\_Triangles}$
11 **return** TC

The brain networks and the Graph-500 instances are dense and large, and therefore, it is hard to store all the enumerated triangles on a PC. We do not require the enumerated triangles during triangle centrality calculation. Therefore, we are interested in testing our parallel triangle centrality algorithm for these instances.

Figure 5.10 shows the speed-up of Algorithm 8 for the brain networks, and Figure 5.11

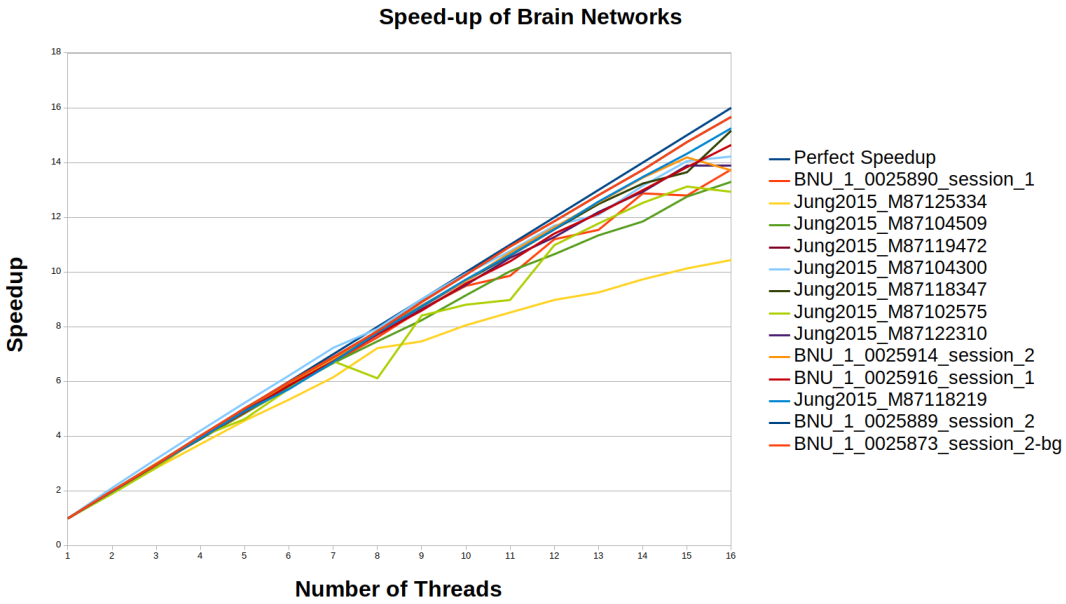


Figure 5.10: Speed-up of brain networks for **ParallelFullCount** algorithm with parallel triangle centrality

presents the speed-up for the Graph-500 networks.

5.5 Conclusion

Network data is usually input as a list of edges which can be preprocessed into a representation such as an adjacency matrix or adjacency list, suitable for algorithmic processing. We have presented a simple, yet flexible scheme based on intersecting edge labels, the intersection matrix, for the representation of and calculation with network data. A new linear algebra-based method exploits this intersection representation for triangle computation – a kernel operation in big data analytics. By using sparse matrix-vector products instead of the memory-intensive matrix-matrix multiplication, our implementation has the capacity to enumerate and extend triangle analysis in graphs so that important information such as triangle vertex and edge degree can be gleaned in a fraction of the time of reference implementation of miniTri on large benchmark instances. The computational results from a set of large-scale synthetic and real-world network instances clearly demonstrate that our basic implementation is efficient and scales well. The two arrays `FDC` and `FN` together constitute

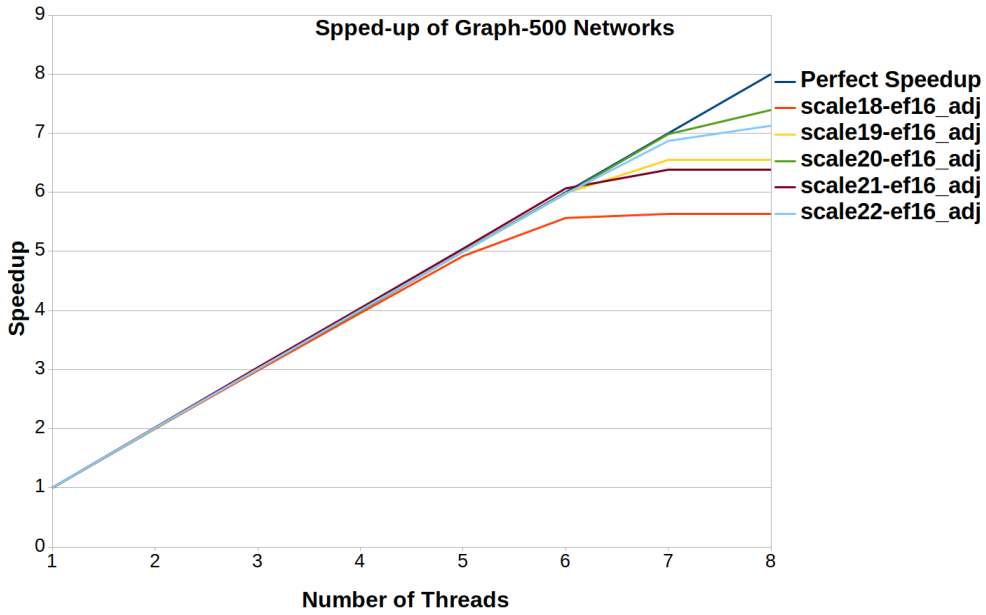


Figure 5.11: Speed-up of Graph-500 networks for **ParallelFullCount** algorithm with parallel triangle centrality

a compact representation of the sparsity pattern of network data, requiring only $n+m$ units of storage. This is incredibly useful in the exchange of network data, with the potential to allow for the computation of many additional intersection matrix-based network analytics such as rank and triangle centrality [9]. The comparative results of our algorithms for ranking and triangle centrality show better scalable results for large instances. A shared memory parallel implementation of `int` algorithm using OpenMP is being developed. We observe reasonable speedups using multiple threads. This algorithm can still be tuned, and cache efficiency is being studied for additional optimizations, exploring temporal and spatial locality to analyze the memory footprint and provide further improvements. A natural extension of the research presented in this paper is to use the intersection representation in *graphlet* counting methods. Similar to the k-count distribution, graphlet frequency distribution (a vector of the frequency of different graphlets in a graph) provides local topological properties of graphs [79].

Chapter 6

Summary and Future Work

6.1 Introduction

For high-quality analysis of large networks, graph enumeration algorithms are considered as a powerful tool. Large real-life, synthetic and dense networks create challenges in network analysis study. This thesis addresses the challenge of representing these large networks. The algorithms considered in this thesis span a wide range of problems and applications: edge clique cover problem, assignment-minimum edge clique cover problem, triangle counting & enumeration problem, k-count, and triangle centrality. These algorithms are all connected to finding important structures within real-life networks. With parallel versions of these graph algorithms, this thesis also focuses on utilizing the threads all modern PCs have. Therefore, one can use this thesis as a guideline for approaching graph enumeration problems with parallel algorithms.

6.2 Results in "Vertex-centric Edge Clique Cover"

In Chapter 3, we showed the connection between large networks and their sparse matrix representation that can be exploited to employ efficient techniques from sparse matrix determination literature in graph algorithms [37, 38].

The edge clique cover problem is recast as a sparse matrix determination problem. The notion of *intersection matrix* provides a unified framework that facilitates the compact representation of graph data and efficient implementation of graph algorithms. The adjacency matrix representation can potentially have many nonzero entries since it is the product of an

intersection matrix with its transpose. We showed that similar to the graph vertex coloring problem, the ECC problem is sensitive to the ordering of the vertices.

We presented test results of "vertex-centric edge clique cover" algorithm, `VO-ECC` for the selected test instances from group DIMACS10, SNAP, and real-life. We compared our algorithm using different vertex ordering methods and showed that for all test instances, the ordered approach produces strictly better clique cover compared with *natural* ordering. Finally, we showed that our algorithm is scalable for large test instances where the comparing algorithm (presented by Gramm et al. [28]) fails to run DIMACS10 and SNAP instances.

6.3 Results in "Edge-centric Edge Clique Cover"

We proposed a compact representation of network data, where the notion of *intersection matrix* provides a unified framework that facilitates the compact representation of graph data and efficient implementation of graph algorithms.

This thesis presented an "edge-centric" minECC method, `EO-ECC` motivated by the works of Bron et al. [7] and E. Tomita et al. [81] for finding clique covers. Then presented comparative results concerning the clique cover size and runtime with the current state-of-the-art algorithm for minECC [17]. For 219 test instances (from DIMACS10, SNAP, Real-World, Small-World, and Erdős-Rényi groups), where the number of edges varies between 170 and 7.6×10^7, our `EO-ECC` algorithm produces smaller or equal size clique covers than the `Conte-Method` [17]. `EO-ECC` is also significantly faster than the `Conte-Method`, and is highly scalable on large problem instances.

A less well-studied but related problem, known as the *Assignment Minimum Edge Clique Cover* arising in computational statistics, is to minimize the number of individual assignments of vertices to cliques. It is not always possible to find assignment-minimum clique coverings by searching through those that are edge-clique-minimum. This thesis addressed this problem and proposed an algorithm `AM-ECC`, where `EO-ECC` with a post-processing step gives the assignment minimum cover calculation.

Finally, the parallel implementation of `EO-ECC` using OpenCilk in this thesis established a guideline for parallel implementation of the edge clique cover problem.

6.4 Results in "Triangle Counting & Enumeration"

The presence of triangles in extensive network data has led to the creation of many metrics to analyze graph characteristics. As such, the ability to count and enumerate these triangles is crucial to applying these metrics and gaining further insights into the underlying composition and distribution of these graphs.

This thesis presented a simple-flexible scheme based on intersecting edge labels, the intersection matrix, to represent and calculate with network data. A new linear algebra-based method exploits this intersection representation for triangle computation – a kernel operation in big data analytics. In our implementation, we used sparse matrix-vector products instead of the memory-intensive matrix-matrix multiplication to enumerate and extend triangle analysis in graphs. Therefore, important information such as triangle vertex and edge degree can be gleaned in a fraction of the time of reference implementation of miniTri on large benchmark instances.

The computational results from large-scale synthetic and real-life network instances demonstrate that our basic implementation is efficient, scales well, and requires $O(n+m)$ space. The presented idea in this thesis is incredibly useful in the exchange of network data, with the potential to allow for the computation of many additional intersection matrix-based network analytics such as rank and triangle centrality [9].

We measured the speedup we gained the two reference miniTri implementations on these real-world instances for our triangle count and enumeration algorithms. However, both miniTri implementations failed to compute triangles for large test instances, such as "flickrEdges" and "Cit-Patents", and also large instances from the brain and Graph500 networks. We reported those failed cases using **N/A**.

Our implemented **fullCount** algorithm is highly scalable, and therefore, we tested this

algorithm with all instances from the brain network and large synthetic test instances from GraphChallenge. The reference implementations failed to calculate results for most of these large instances while we ran those for seven days with 900GB memory in a high-performance Computing system (Graham cluster) at Compute Canada.

Calculating k-count requires huge space to store enumerated triangles, and for the dense brain networks, it is challenging. Therefore, to test the scalability and runtime of these dense graphs, we avoid calculating k-count and only calculate triangles, triangle vertex degree, and triangle edge degree.

In this thesis, we also implemented triangle centrality using our **fullCount** algorithm. We compared our result with Li and Bader's [51] SuiteSparse GraphBLAS implementation.

Our method calculated triangle centrality within a fraction of a second for all the instances. Our triangle centrality calculation requires triangle counting and triangle vertex & edge degree calculation. Therefore, if we consider the running time for our **fullCount** algorithm with our triangle centrality calculation, still we get better performance except for one instance `as-Skitter`. These results showed a prospect of using our triangle centrality calculation method for large instances.

Finally, we presented the parallel implementation of our intersection algorithm, **fullCount**. We parallelized our basic triangle counting algorithm, **fullCount** algorithm, k-count, and triangle centrality calculation method. All of this implementation showed reasonable speedup for using multiple threads.

6.5 Future work

- Our vertex-ordered ECC (VO-ECC) algorithm selects an unprocessed vertex, and then it examines all the existing cliques to find whether we can include this processing vertex into an existing clique or not. Therefore, we required a two-dimensional array to store and grow the clique cover. In a future implementation, we will try to use the intersection-based matrix to store the cover to reduce the space complexity.

- There are two important functions in our EO-ECC algorithm: *FindNeighbors*, and *FindCommonNeighbors*. We have implemented parallelized version of these functions. Therefore, one future direction can parallelize the full EO-ECC algorithm.

- We have developed a parallelized version of our triangle counting and enumeration algorithm. This implementation has optimistic preliminary results. This algorithm can still be tuned, and cache efficiency is being studied for additional optimizations, exploring temporal and spatial locality to analyze the memory footprint and provide further improvements.

6.6 Conclusion

Analyzing characteristics in networked data, such as graphs that can yield important information on the modelled structure, is challenging due to their linked nature and size. For classification, clustering, and knowledge discovery, analyzing subgraphs is helpful to get a deeper understanding of the data. This thesis proposed using a compact network data representation based on sparse matrix data structures. We considered the enumeration of subgraphs (edge clique cover problem) with some ordering schemes. We used the linear algebraic approach to implement graph algorithms for counting triangles, triangle enumeration, the k-count, and triangle centrality calculation. This thesis presented both serial and parallel algorithms for solving these graph problems in analyzing social and large complex networks.

Bibliography

[1] W. M. Abdullah, S. Hossain, and M. A. Khan. Covering large complex networks by cliques—a sparse matrix approach. In D. Marc Kilgour, Herb Kunze, Roman Makarov, Roderick Melnik, and Xu Wang, editors, *Recent Developments in Mathematical, Statistical and Computational Sciences*, pages 117–127, Cham, 2021. Springer International Publishing.

[2] Mohammad Al Hasan and Vachik S Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.

[3] Ghadeer Alabandi, Evan Powers, and Martin Burtscher. Increasing the parallelism of graph coloring via shortcutting. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–275, 2020.

[4] L Becchetti, P Boldi, and C Castillo. Efficient algorithms for large-scale local triangle counting. In *ACM Trans Knowl Discovery Data*, pages 1–28, 2010.

[5] Claude Berge. *Graphs and hypergraphs*. North-Holland Pub. Co., 1973.

[6] M Blanchette, E Kim, and A Vetta. Clique cover on sparse networks. *In 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics*, pages 93–102, 2012.

[7] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[8] Paul Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16(3):157–166, 2017.

[9] Paul Burkhardt. Triangle centrality. *ArXiv*, abs/2105.00110, 2021.

[10] Charles Cable, Kathryn F Jones, J Richard Lundgren, and Suzanne Seager. Niche graphs. *Discrete applied mathematics*, 23(3):231–241, 1989.

[11] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594, 2012.

[12] Graham cluster. Compute Canada high power computing resource. `https://docs.computecanada.ca/wiki/Graham`, 2021. [Online; accessed 28-November-2021].

[13] Joel E. Cohen. Interval graphs and food webs: a finding and a problem. *RAND Corporation Document 17696-PR*, 1968.

[14] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 16(3.1), 2008.

[15] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the gpu. In *GPU Technology Conference*, pages 1–10, 2012.

[16] Thomas F Coleman and Jorge J Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[17] Alessio Conte, Roberto Grossi, and Andrea Marino. Large-scale clique cover of real-world networks. *Information and Computation*, 270:104464, 2020.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[19] Marek Cygan, Marcin Pilipczuk, and Michał Pilipczuk. Known algorithms for edge clique cover are probably optimal. *SIAM Journal on Computing*, 45(1):67–83, 2016.

[20] T. Davis and Y. Hu. Suitesparse matrix collection. https://sparse.tamu.edu/. Accessed: 2019-10-02.

[21] Daniel M Ennis, Benoit Rousseau, and John M Ennis. *Tools and Applications of Sensory and Consumer Science: 59 Technical Report Scenarios Based on Real-life Problems*. Institute for Perception, 2014.

[22] JM Ennis and DM Ennis. Efficient Representation of Pairwise Sensory Information. *IFPress*, 15(3):3–4, 2012.

[23] John M Ennis, Charles M Fayle, and Daniel M Ennis. Assignment-minimum clique coverings. *Journal of Experimental Algorithmics (JEA)*, 17:1–1, 2012.

[24] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation*, pages 403–414. Springer, 2010.

[25] David Eppstein and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*, pages 364–375. Springer, 2011.

[26] Paul Erdös, Adolph W Goodman, and Louis Pósa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18:106–112, 1966.

[27] J. Gramm, J. Guo, F. Huffner, and R. Niedermeier. Data reduction, Exact and Heuristic Algorithms for Clique Cover. *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM*, pages 86–94, 2006.

[28] J. Gramm, J. Guo, F. Huffner, and R. Niedermeier. Data reduction and exact algorithms for clique cover. *Journal of Experimental Algorithmics (JEA)*, 13:2–15, 2009.

[29] J. Gramm, J. Guo, F. Huffner, R. Niedermeier, H. Piepho, and R. Schmid. Algorithms for Compact Letter Displays: Comparison and Evaluation. *Computational Statistics & Data Analysis*, 52:725–736, 2007.

[30] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *Journal of Experimental Algorithmics (JEA)*, 13:2–2, 2009.

[31] András Gyárfás. A simple lower bound on edge coverings by cliques. *Discrete Mathematics*, 85(1):103–104, 1990.

[32] M. Hasan, S. Hossain, A. I. Khan, N. H. Mithila, and A. H. Suny. DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices. In: G.M. Greuel, T. Koch, P. Paule, A. Sommese and Editors. ICMS2016. *Springer International Publishing Switzerland*, pages 425–434, 2016.

[33] Mahmudul Hasan, Shahadat Hossain, Ahamad Imtiaz Khan, Nasrin Hakim Mithila, and Ashraful Huq Suny. DSJM: a software toolkit for direct determination of sparse Jacobian matrices. In *International Congress on Mathematical Software*, pages 275 – 283. Springer, 2016.

[34] William Hasenplaugh, Tim Kaler, Tao B Schardl, and Charles E Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 166–177, 2014.

[35] S. Hossain and A. I. Khan. Exact Coloring of Sparse Matrices. In: D.M. Kilgour et al. (eds.) Recent Advances in Mathematical and Statistical Methods. *Springer Proceedings in Mathematics and Statistics, Springer Nature Switzerland AG*, 259:23–36, 2018.

[36] S. Hossain and A. H. Suny. Determination of Large Sparse Derivative Matrices: Structural: Orthogonality and Structural Degeneracy. *In: B. Randerath, H. Roglin, B. Peis, O. Schaudt, R. Schrader, F. Vallentin and V. Weil. 15th Cologne-Twente Workshop on Graphs & Combinatorial Optimization, Cologne, Germany*, pages 83–87, 2017.

[37] Shahadat Hossain and Trond Steihaug. Graph models and their efficient implementation for sparse jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013.

[38] Shahadat Hossain and Trond Steihaug. Optimal direct determination of sparse jacobian matrices. *Optimization Methods and Software*, 28(6):1218–1232, 2013.

[39] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[40] O. James. Contentment in graph theory: covering graphs with cliques. *Indagationes Mathematicae (Proceedings)*, 80(5), 1977.

[41] Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[42] E. Kellerman. Determination of keyword conflict. *IBM Technical Disclosure Bulletin*, 16(2):544–546, 1973.

[43] J. Kepner and J. Gilbert. Graph Algorithms in the Language of Linear Algebra, Society for Industrial and Applied Mathematics. *Philadelphia, PA, USA*, 2011.

[44] J. Kepner and H. Jananthan. Mathematics of big data: Spreadsheets, databases, matrices, and graphs. *MIT Press*, 2018.

[45] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[46] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[47] LT Kou, LJ Stockmeyer, and CK Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Communications of the ACM*, 21(2):135–139, 1978.

[48] VP Kozyrev and SV Yushmanov. Representations of graphs and networks (coding, layouts and embeddings). *Journal of Soviet Mathematics*, 61(3):2152–2194, 1992.

[49] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. Accessed: 2019-10-02.

[50] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[51] Fuhuan Li and David A Bader. A graphblas implementation of triangle centrality. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2. IEEE, 2021.

[52] László Lovász. On covering of graphs. In *Theory of Graphs (Proc. Colloq., Tihany, 1966)*, pages 231–236. Academic Press New York, 1968.

[53] Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru Popovici, Franz Franchetti, and Scott McMillan. First look: Linear algebra-based triangle counting without matrix multiplication. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017.

[54] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.

[55] Sean McGuinness and Rolf Rees. On the number of distinct minimal clique partitions and clique covers of a line graph. *Discrete mathematics*, 83(1):49–62, 1990.

[56] R Milo, S Shen-Orr, and S Itzkovitz. Network motifs: simple building blocks of complex network. *Science*, pages 824–827, 2002.

[57] Egbert Mujuni and Frances Rosamond. Parameterized complexity of the clique partition problem. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77*, pages 75–78, 2008.

[58] Matthias Müller-Hannemann and Stefan Schirra. *Algorithm Engineering*. Springer, 2001.

[59] MA Nestrud, JM Ennis, CM Fayle, DM Ennis, and HT Lawless. Validating a graph theoretic screening approach to food item combinations. *Journal of sensory studies*, 26(5):331–338, 2011.

[60] Robert J Opsut. On the computation of the competition number of a graph. *SIAM Journal on Algebraic Discrete Methods*, 3(4):420–428, 1982.

[61] James Orlin. Contentment in graph theory: covering graphs with cliques. In *Indagationes Mathematicae (Proceedings)*, volume 80(5), pages 406–424. Elsevier, 1977.

[62] G Palla, I Dereny, I Frakas, and T Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, pages 814–818, 2005.

[63] Svatopluk Poljak, Vojtěch Rödl, and Daniel TURZiK. Complexity of representation of graphs by set systems. *Discrete Applied Mathematics*, 3(4):301–312, 1981.

[64] Erich Prisner. Clique covering and clique partition in generalizations of line graphs. *Discrete applied mathematics*, 56(1):93–98, 1995.

[65] Norman J Pullman. Clique coverings of graphs—a survey. *Combinatorial Mathematics X*, pages 72–85, 1983.

[66] Fred S Roberts. Food webs, competition graphs, and the boxicity of ecological phase space. In *Theory and Applications of Graphs*, pages 477–490. Springer, 1978.

[67] Fred S Roberts. Applications of edge coverings by cliques. *Discrete applied mathematics*, 10(1):93–109, 1985.

[68] Marcos Okamura Rodrigues. Fast constructive and improvement heuristics for edge clique covering. *Discrete Optimization*, 39:100628, 2021.

[69] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[70] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[71] Ryan A Rossi, David F Gleich, Assefaw H Gebremedhin, and Md Mostofa Ali Patwary. Fast maximum clique algorithms for large graphs. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 365–366, 2014.

[72] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Static graph challenge: Subgraph isomorphism. http://graphchallenge.mit.edu/data-sets, 2017. Accessed: 2021-07-09.

[73] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Graphchallenge.org triangle counting performance, 2020.

[74] Yoshio Sano. A generalization of opsut's lower bounds for the competition number of a graph. *Graphs and Combinatorics*, 29(5):1543–1547, 2013.

[75] Tao B Schardl, I-Ting Angelina Lee, and Charles E Leiserson. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 351–353, 2018.

[76] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.

[77] Nandini Singhal, Sathya Peri, and Subrahmanyam Kalyanasundaram. Practical multithreaded graph coloring algorithms for shared memory architecture. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–7, 2017.

[78] Peter J Slater. A note on pseudointersection graphs. *J. Res. Nat. Bur. Standards B*, 80:441–445, 1976.

[79] Edward Szpilrajn-Marczewski. A translation of sur deux propriétés des classes d'ensembles by. *Fund. Math*, 33:303–307, 1945.

[80] G Tinhofer. Generating graphs uniformly at random. In *Computational graph theory*, pages 235–255. Springer, 1990.

[81] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical computer science*, 363(1):28–42, 2006.

[82] S Wasserman and K Faust. Social network analysis: Methods and applications. *Cambridge university press*, 1994.

[83] Michael M Wolf, Jonathan W Berry, and Dylan T Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2015.

[84] Elena Zotenko, Katia S Guimarães, Raja Jothi, and Teresa M Przytycka. Decomposition of overlapping protein complexes: A graph theoretical method for analyzing static and dynamic protein associations. In *Systems Biology and Regulatory Genomics*, pages 23–38. Springer, 2005.

Appendix A

Table 1: Test Results (number of cliques) for Erdos-Renyi and Small-World Graphs

Graph			Number of cliques			
Name	*m*	*n*	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
er_n2	800	100	374	365	367	368
er_1p5n2	1200	150	667	640	645	643
er_2n2	1600	200	975	954	959	957
er_2p5n2	2000	250	1274	1255	1256	1257
er_3n2	2400	300	1596	1580	1584	1582
er_3p5n2	2800	350	1993	1984	1987	1985
er_4n2	3200	400	2328	2324	2324	2322
er_4p5n2	3600	450	2691	2678	2678	2680
er_5n2	4000	500	3024	3016	3016	3016
er_5p5n2	4400	550	3399	3395	3395	3396
er_6n2	4800	600	3808	3804	3804	3804
er_6p5n2	5200	650	4215	4208	4209	4209
er_7n2	5600	700	4588	4587	4587	4587
er_7p5n2	6000	750	4914	4911	4911	4911
er_8n2	6400	800	5305	5302	5302	5302
er_8p5n2	6800	850	5708	5707	5707	5707
er_9n2	7200	900	6116	6115	6115	6115
er_9p5n2	7600	950	6424	6422	6422	6422
er_n3	8000	1000	6821	6821	6821	6821
er_1p5n3	12000	1500	10777	10777	10777	10777
er_2n3	16000	2000	14782	14782	14782	14782
er_2p5n3	20000	2500	18635	18634	18634	18634
er_3n3	24000	3000	22652	22652	22652	22652
er_3p5n3	28000	3500	26712	26712	26712	26712
er_4n3	32000	4000	30705	30704	30704	30704
er_4p5n3	36000	4500	34720	34720	34720	34720
er_5n3	40000	5000	38647	38647	38647	38647
er_5p5n3	44000	5500	42646	42646	42646	42646
er_6n3	48000	6000	46558	46558	46558	46558
er_6p5n3	52000	6500	50681	50681	50681	50681
er_7n3	56000	7000	54586	54586	54586	54586
er_7p5n3	60000	7500	58615	58615	58615	58615
er_8n3	64000	8000	62699	62699	62699	62699
er_8p5n3	68000	8500	66620	66620	66620	66620
er_9n3	72000	9000	70681	70681	70681	70681
er_9p5n3	76000	9500	74710	74710	74710	74710

Table 1 – *Continued on next page*

Table 1 – *Continued from previous page*

Graph			Number of cliques			
Name	*m*	*n*	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
er_n4	80000	10000	78650	78650	78650	78650
er_1p5n4	120000	15000	118693	118693	118693	118693
er_2n4	160000	20000	158649	158649	158649	158649
er_2p5n4	200000	25000	198593	198593	198593	198593
er_3n4	240000	30000	238669	238669	238669	238669
er_3p5n4	280000	35000	278675	278675	278675	278675
er_4n4	320000	40000	318660	318660	318660	318660
er_4p5n4	360000	45000	358654	358654	358654	358654
er_5n4	400000	50000	398648	398648	398648	398648
er_5p5n4	440000	55000	438639	438639	438639	438639
er_6n4	480000	60000	478666	478666	478666	478666
er_6p5n4	520000	65000	518511	518511	518511	518511
er_7n4	560000	70000	558667	558667	558667	558667
er_7p5n4	600000	75000	598576	598576	598576	598576
er_8n4	640000	80000	638619	638619	638619	638619
er_8p5n4	680000	85000	678725	678725	678725	678725
er_9n4	720000	90000	718613	718613	718613	718613
er_9p5n4	760000	95000	758554	758554	758554	758554
er_n5	800000	100000	798737	798737	798737	798737
er_1p5n5	1200000	150000	1198666	1198666	1198666	1198666
er_2n5	1600000	200000	1598603	1598603	1598603	1598603
er_2p5n5	2000000	250000	1998582	1998582	1998582	1998582
er_3n5	2400000	300000	2398561	2398561	2398561	2398561
er_3p5n5	2800000	350000	2798653	2798653	2798653	2798653
er_4n5	3200000	400000	3198576	3198576	3198576	3198576
er_4p5n5	3600000	450000	3598533	3598533	3598533	3598533
er_5n5	4000000	500000	3998588	3998588	3998588	3998588
er_5p5n5	4400000	550000	4398603	4398603	4398603	4398603
er_6n5	4800000	600000	4798718	4798718	4798718	4798718
er_6p5n5	5200000	650000	5198570	5198570	5198570	5198570
er_7n5	5600000	700000	5598612	5598612	5598612	5598612
er_7p5n5	6000000	750000	5998716	5998716	5998716	5998716
er_8n5	6400000	800000	6398614	6398614	6398614	6398614
er_8p5n5	6800000	850000	6798625	6798625	6798625	6798625
er_9n5	7200000	900000	7198633	7198633	7198633	7198633
er_9p5n5	7600000	950000	7598639	7598639	7598639	7598639
er_n6	8000000	1000000	7998545	7998545	7998545	7998545
er_1p5n6	12000000	1500000	11998646	11998646	11998646	11998646

Table 1 – *Continued on next page*

Table 1 – *Continued from previous page*

Graph			Number of cliques			
Name	m	n	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
er_2n6	16000000	2000000	15998621	15998621	15998621	15998621
er_2p5n6	20000000	2500000	19998715	19998715	19998715	19998715
er_3n6	24000000	3000000	23998646	23998646	23998646	23998646
er_3p5n6	28000000	3500000	27998674	27998674	27998674	27998674
er_4n6	32000000	4000000	31998755	31998755	31998755	31998755
er_4p5n6	36000000	4500000	35998669	35998669	35998669	35998669
er_5n6	40000000	5000000	39998710	39998710	39998710	39998710
er_5p5n6	44000000	5500000	43998661	43998661	43998661	43998661
er_6n6	48000000	6000000	47998612	47998612	47998612	47998612
er_6p5n6	52000000	6500000	51998699	51998699	51998699	51998699
er_7n6	56000000	7000000	55998696	55998696	55998696	55998696
er_7p5n6	60000000	7500000	59998656	59998656	59998656	59998656
er_8n6	64000000	8000000	63998661	63998661	63998661	63998661
er_8p5n6	68000000	8500000	67998726	67998726	67998726	67998726
er_9n6	72000000	9000000	71998747	71998747	71998747	71998747
er_9p5n6	76000000	9500000	N/A	75998685	75998685	75998685
er_2n2_dup	1600	200	941	930	935	929
er_4n2_dup	3200	400	2350	2342	2341	2341
er_7n2_dup	5600	700	4555	4554	4554	4554
er_9n2_dup	7200	900	6135	6133	6133	6133
er_2n3_dup	16000	2000	14711	14711	14711	14711
er_4n3_dup	32000	4000	30791	30791	30791	30791
er_7n3_dup	56000	7000	54724	54724	54724	54724
er_9n3_dup	72000	9000	70661	70661	70661	70661
er_2n4_dup	160000	20000	158616	158616	158616	158616
er_4n4_dup	320000	40000	318764	318764	318764	318764
er_7n4_dup	560000	70000	558540	558540	558540	558540
er_9n4_dup	720000	90000	718638	718638	718638	718638
er_2n5_dup	1600000	200000	1598668	1598668	1598668	1598668
er_4n5_dup	3200000	400000	3198716	3198716	3198716	3198716
er_7n5_dup	5600000	700000	5598523	5598523	5598523	5598523
er_9n5_dup	7200000	900000	7198570	7198570	7198570	7198570
er_3n6_dup	24000000	3000000	23998533	23998533	23998533	23998533
er_5n6_dup	40000000	5000000	39998674	39998674	39998674	39998674
er_7n6_dup	56000000	7000000	55998702	55998702	55998702	55998702
er_9n6_dup	72000000	9000000	71998707	71998707	71998707	71998707
sw_n2	777	100	251	248	249	246
sw_1p5n2	1184	150	427	427	433	426

Table 1 – *Continued on next page*

Table 1 – *Continued from previous page*

Graph			Number of cliques			
Name	m	n	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
sw_2n2	1580	200	599	584	591	589
sw_2p5n2	1977	250	765	755	764	755
sw_3n2	2382	300	959	948	943	945
sw_3p5n2	2776	350	1152	1128	1136	1129
sw_4n2	3175	400	1323	1309	1310	1309
sw_4p5n2	3575	450	1566	1541	1538	1539
sw_5n2	3974	500	1693	1671	1691	1677
sw_5p5n2	4385	550	1961	1929	1954	1933
sw_6n2	4778	600	2084	2056	2068	2051
sw_6p5n2	5173	650	2295	2257	2276	2259
sw_7n2	5577	700	2472	2440	2450	2441
sw_7p5n2	5981	750	2673	2635	2640	2630
sw_8n2	6373	800	2905	2863	2893	2875
sw_8p5n2	6780	850	3100	3061	3074	3061
sw_9n2	7182	900	3280	3225	3226	3223
sw_9p5n2	7579	950	3438	3371	3398	3387
sw_n3	7981	1000	3598	3528	3562	3554
sw_1p5n3	11972	1500	5421	5317	5358	5328
sw_2n3	15974	2000	7312	7232	7261	7218
sw_2p5n3	19974	2500	9303	9134	9202	9182
sw_3n3	23975	3000	11227	11037	11105	11041
sw_3p5n3	27979	3500	13112	12923	13006	12946
sw_4n3	31985	4000	15040	14839	14892	14814
sw_4p5n3	35973	4500	16866	16655	16753	16671
sw_5n3	39982	5000	18899	18614	18716	18640
sw_5p5n3	43973	5500	20857	20573	20658	20603
sw_6n3	47983	6000	22760	22439	22602	22476
sw_6p5n3	51977	6500	24620	24294	24406	24295
sw_7n3	55981	7000	26701	26351	26475	26344
sw_7p5n3	59973	7500	28340	27941	28084	27938
sw_8n3	63983	8000	30132	29679	29830	29675
sw_8p5n3	67973	8500	32349	31861	32060	31894
sw_9n3	71970	9000	34496	34043	34185	34000
sw_9p5n3	75981	9500	36045	35537	35707	35548
sw_n4	79981	10000	38297	37728	37922	37738
sw_1p5n4	119965	15000	57200	56427	56733	56437
sw_2n4	159974	20000	76420	75311	75675	75300
sw_2p5n4	199978	25000	95423	93979	94489	94012

Table 1 – *Continued on next page*

Table 1 – *Continued from previous page*

Graph			Number of cliques			
Name	*m*	*n*	Conte-Method	EO-ECC-D	EO-ECC-L	EO-ECC-I
sw_3n4	239983	30000	114310	112596	113186	112706
sw_3p5n4	279973	35000	133609	131469	132384	131714
sw_4n4	319977	40000	153196	150879	151860	151059
sw_4p5n4	359982	45000	172231	169603	170584	169744
sw_5n4	399977	50000	191447	188535	189601	188721
sw_5p5n4	439973	55000	210817	207723	208965	207896
sw_6n4	479977	60000	230235	226878	228308	227069
sw_6p5n4	519976	65000	249033	245310	246597	245467
sw_7n4	559981	70000	268415	264532	266167	264758
sw_7p5n4	599973	75000	286940	282570	284349	282897
sw_8n4	639982	80000	305777	301354	302920	301448
sw_8p5n4	679974	85000	324852	319896	321770	320072
sw_9n4	719976	90000	344627	339563	341412	339867
sw_9p5n4	759979	95000	363134	357848	359811	358243
sw_n5	799982	100000	382413	376638	378904	376915
sw_1p5n5	1199984	150000	573387	564891	568221	565304
sw_2n5	1599982	200000	764882	753682	757845	754086
sw_2p5n5	1999981	250000	956541	942263	947438	942795
sw_3n5	2399977	300000	1147827	1130875	1137471	1131681
sw_3p5n5	2799977	350000	1340075	1320374	1327843	1321250
sw_4n5	3199982	400000	1530792	1508066	1516658	1509066
sw_4p5n5	3599981	450000	1722783	1697178	1706735	1698305
sw_5n5	3999977	500000	1913519	1885375	1896492	1886862
sw_5p5n5	4399981	550000	2105795	2074576	2086190	2076190
sw_6n5	4799984	600000	2298734	2265206	2277675	2266410
sw_6p5n5	5199990	650000	2488676	2451375	2465513	2453064
sw_7n5	5599975	700000	2678465	2638847	2653956	2640735
sw_7p5n5	5999972	750000	2870722	2828343	2843773	2830134
sw_8n5	6399973	800000	3063567	3017993	3034988	3019657
sw_8p5n5	6799966	850000	3252942	3204666	3222967	3207088
sw_9n5	7199983	900000	3446088	3395083	3414789	3397958
sw_9p5n5	7599982	950000	3638065	3584306	3604182	3586335